图灵程序设计丛书

Python项目开发实战

第2版

【日】日本BePROUD股份有限公司 著 支鹏浩 译

会写代码≠能做好项目！

- 建立有序生产环境
- 迅速融入开发团队
- 高效处理项目问题

网罗Python项目开发中的流程
让你的编程事半功倍

Python项目与封装/团队开发环境/问题驱动开发/源码管理（Mercurial）
Jenkins持续集成（CI）/环境搭建与部署的自动化（Ansible）/Django框架……

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Python项目开发实战

第2版

【日】日本BePROUD股份有限公司 著 支鹏浩 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

Python项目开发实战 / 日本BePROUD股份有限公司著；
支鹏浩译. -- 2版. -- 北京：人民邮电出版社，
2017.1
（图灵程序设计丛书）
ISBN 978-7-115-43856-0

Ⅰ. ①P… Ⅱ. ①日… ②支… Ⅲ. ①软件工具－程序
设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字（2016）第254060号

内 容 提 要

本书来自真正的开发现场，是BePROUD公司众多极客在真实项目中的经验总结和智慧结晶。作者从Python的环境搭建开始讲起，介绍了Web应用的开发方法、项目管理及审查、测试与高效部署、服务器调试等内容，尽可能网罗了Python项目开发流程中的方方面面，有助于开发者建立有序生产环境，提高开发效率，让编程事半功倍。此外，在本书中Python仅仅是一个载体，很多知识点在非Python下也适用。

本书适合有一定基础的Python开发者，以及使用PHP或Ruby进行开发的读者阅读。

◆ 著　　　[日]日本BePROUD股份有限公司
译　　　支鹏浩
责任编辑　傅志红
执行编辑　高宇涵　侯秀娟
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：27.5
字数：650千字　　　　2017年1月第1版
印数：1－4 000册　　　2017年1月北京第1次印刷
著作权合同登记号　图字：01-2015-6980号

定价：79.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第8052号

引　言

迄今为止，BePROUD 公司已使用 Python 开发了诸多项目。我们之所以撰写本书，是为了与各位读者分享我们在实践中总结出的一些技巧。

同时，鉴于最近公司员工数量增长，我们把在 BePROUD 工作所需的知识也写入了本书，以便新的公司成员能尽快熟悉工作。

因此本书从搭建工作环境开始讲起，逐步涉及 Web 应用的开发、项目管理及审查、测试代码的编写与高效部署、服务器调试等方面，网罗了 Python 项目开发工作中的一系列流程。书名中的“实战”一词就包含了“工作”的意思。

书中所写的技巧主要源于我们的 Python2 开发经验。也正因为如此，本书将以 Python2 为例进行讲解。如今新的开发项目已经在使用 Python3，这些技巧转移到 Python3 上理应同样适用。

进入正题之前，先来聊聊我们的日常思路。

◉ 极客 / 书虫常伴身边的公司

BePROUD 里不乏极客和书虫们。在这里，很多人对特定领域的了解程度能吓掉你的下巴。

在这里，人们一旦发现感兴趣的事，就会拿出私人时间来学习、实践。要知道，极客和书虫们不会为这种事情吝啬时间。

正如人们印象中的那样，极客和书虫们大多有些怪癖，但 BePROUD 的员工都具备下列共识。

- **希望能不做不想做的事**
- **希望学会好的方法并付诸实践**
- **希望工作时有个好心情**

◉ 希望能不做不想做的事

在工作中，重复单调的作业是一种极其无趣的事，因此能一次办完的事谁都不想去办两次。另外，大家也都讨厌工序复杂、容易出错的工作。所以要开动脑筋，把复杂的工序简单化，同时尽量减少出错的机会。

◉ 希望学会好的方法并付诸实践

世界上有许多公认的好方法、新思路和新技巧，我们要勇于尝试，学习它们并付诸实践。

使用好的方法必然能帮助我们削减不想做的工作。不过，方法的好坏不能人云亦云，我们必须选出对自己真正有帮助的方法，然后再将所学方法应用到实际业务当中。

◉ 希望工作时有个好心情

现在，我们学会了优秀的方法、削减了繁杂的工作，之后自然希望带着好心情去工作。此时不妨给 Skype 做个好玩的 bot，或者在下班后找个会议室搞一场妙趣横生的快速演讲。我们希望大家能在保质保量完成工作的同时有个好心情，而不是只把公司当作工作的场所。这是我们的理念。

本书的内容全部基于事实，都是 BePROUD 员工实际尝试、实践过的。我们希望给各位提供一些能实际应用且行之有效的知识，而不是让各位去死记硬背一大堆晦涩难懂的概念。我们很愿意看到本书的知识能对各位有所帮助，愿各位能在工作中有个好心情。

◉ 谢辞

本书在编撰过程中承蒙多名 IT 业界高人指点：寺田学（@terapyon）、金子望、关根裕纪（@checkpoint）、畠弥峰（@flag_boy）、小坂健二朗（@inoshiro）、筒井隆次（@ryu22e）、永井孝（@ngi644）、中西直树（@nk24）、尾曾越雅文、柴田正明（@_mshibata）、真幡康德（@mahata）、中石宜亨（@eiryplus）。各位在百忙之中仍担起审校工作，慷慨赐教，我们在此表示由衷的感谢。此外还有 BePROUD 公司的 haru、altnight、masaya、crohaco、nakagami、yyyk，感谢几位一边处理着公司内繁忙的开发工作，一边见缝插针地为本书进行审校。

最后感谢各位未能在著者处署名的 BePROUD 员工。如果没有各位员工长期以来的切磋琢磨，这本书永远不会问世。

至此，希望这本集诸人之力编撰出来的书，能为 IT 业界出一份绵薄之力。

全体执笔者

2015 年 1 月

◭ 本书网址

http://www.ituring.com.cn/book/1719

◭ 本书介绍的软件版本和 URL 均为截止到 2015 年 1 月底的最新信息，当前可能已发生变更。

前　言

◉ 本书涉及的内容

本书分为 4 个部分，共 15 章。

第 1 部分“Python 开发入门”的重点将放在个人开发。内容涵盖 Python 开发过程中必不可少的工具的安装（第 1 章），简单的 Web 应用开发（第 2 章）以及 Python 项目的结构与包的创建（第 3 章）。

第 2 部分“团队开发的周期”将为各位说明多人团队开发的相关问题。这部分将重点介绍团队高效开发过程中不可或缺的技术和技巧，内容涵盖团队开发前的环境调整（第 4 章）、项目管理与审查（第 5 章）、源码管理（第 6 章）、文档（第 7 章）、模块设计与单元测试（第 8 章）、封装及其运用（第 9 章）、持续集成（第 10 章）等。

第 3 部分“服务公开”将向各位讲解如何搭建与运用正式环境公开 Web 服务（第 11 章），此外就是有关性能调节的一些方法（第 12 章）。

第 4 部分“加速开发的技巧”可以说是加速开发的一些小贴士。例如将测试的概念导入整个开发流程以加快项目进度（第 13 章），Django 的基础及其进阶性、实践性的用法（第 14 章），Python 的辅助模块（第 15 章）等。

◉ 阅读本书前的准备

环境及版本

- OS: Ubuntu-14.04
- Python: 2.7.6
- Bash: 4.3
- 从第 2 章起，如无特别说明，则运行环境皆由 virtualenv 搭建。

关于 OS

实体机使用 Windows/OS X/Linux，服务器的测试环境使用虚拟机上的 Ubuntu。

Python 的官方手册

```
https://docs.python.org/2.7/
```

我们仅对 Python 官方手册中的内容做最低限度的介绍，部分说明会被省略。因此建议各位

手边时常准备一份参考手册以便阅读。

Python 的官方教程非常适用于学习 Python 的基本安装流程、语法、术语、类以及模块。本书将以各位看过这份教程为前提进行讲解。

Unix/Linux 的一般命令操作

本书虽以 Ubuntu Linux 为前提讲解，但书中不对 Ubuntu Linux 的基本命令操作进行说明。

关于 PyPI（Python Package Index）

PyPI[①] 是一个集中管理包的网站，pip 等自动包安装工具会用到它。本书使用的包也来自 PyPI。

关于敏捷过程与极限编程

本书并不对敏捷过程（Agile Process）和极限编程（ExtremeProgramming）做单独的说明。如今在许多书籍和网站上都能找到这二者的介绍，感兴趣的读者可以去读一读。

本书面向的人群

- 希望改善个人开发环境的人
- 希望改善团队开发的人
- 想学习工作中可使用的 Python 技巧的人
- 新加入 BePROUD 公司项目的成员

◉ 注意

- 本书基于作者本人的调查结果而成。
- 我们在加工本书时力求完美。不过若您发现本书存在不足和错误、漏记等问题，请书面联系出版方。
- 对于因本书内容运用不当而导致的结果及其影响，无论是否因上述两项内容引起，我们均不负责，请知悉。
- 未获得出版方书面许可不得全部或部分复制本书。

◉ 商标等

- 本书已省略 ™ ® © 等符号。
- Python 徽标是 the Python Software Foundation 的商标。
- Django 和 Django 徽标是 Django Software Foundation 的商标。
- Google App Engine 是 Google Inc. 的商标。
- Jenkins 是 SOFTWARE IN PUBLIC INTEREST, INC. 的商标。
- nginx 是 Nginx Software Inc. 的商标。
- VirtualBox 是 ORACLE AMERICA, INC. 的商标。
- Ubuntu 是 Canonical Limited 的商标。
- 此外，公司名和商品名、系统名一般为各开发者的注册商标。
- 本书注册商标中还使用了普遍使用的通用名。

① `https://pypi.python.org/pypi`

目　录

第 4 部分　加速开发的技巧　315

第 1 部分

Python 开发入门

在第 1 部分中，我们将搭建 Python 的基本环境，并进行简单的 Web 应用开发。然后根据已开发出的 Web 应用进一步优化环境，继续开发流程，直至其可以公开给一般用户使用。

第 1 章　Python 入门

各位在学习新技术或新编程语言时，是否对准备工作发过愁呢？往往学习还没有正式开始，就先在准备工作上迷失了方向。好不容易硬着头皮开始准备，却发现安装完一个软件之后又不知道该干什么了。最后自以为准备完毕兴冲冲地要开工时，才注意到应该装好的东西并没有正确安装。到头来，大把的时间花在了准备阶段上，再无心情去学习了。类似这种情况不知道各位遇到过没有。

搭建 Python 开发环境时要考虑 OS 与版本等诸多组合，所以这个过程很难保证一帆风顺。独立开发者，尤其是自学成才的开发者们，大多是以网页或书籍上的信息作为参考，然后用自己独有的方法进行搭建。但即便如此，其中有些共通点还是需要了解的。

第 1 章中，我们将按部就班地了解对个人开发者来说共通的环境搭建顺序，让初学者也能顺利搭建环境。

因此，我们将在第 1 章中对下列项目进行重点学习。其中有部分内容涉及虚拟机，所以我们将学习时使用的本地实体机的 OS 称为“主 OS”，虚拟机的 OS 称为“客 OS”。已经自己搭建好 Python 开发环境的读者可以跳过本章。

- 安装 Python
- 安装 Mercurial
- 关于编辑器以及开发辅助工具

1.1　安装 Python

本节将带领各位安装一些便于在 Ubuntu 上使用 Python 进行开发的工具和包。

NOTE

本书以 Ubuntu 14.04（Server 版）作为 Python 开发环境的 OS。另外，我们使用 OracleVMVirtualBox 承载客 OS。搭建环境的相关内容收录在附录 A 以及附录 B 中，初学者请先参考附录再阅读以下内容。

1.1.1　安装 deb 包

Ubuntu 可以用 `apt-get` 命令管理包。我们先来更新所有包，同时安装一些 Python 开发所需的包。

LIST 1.1　deb 包的更新、升级

```
$ sudo apt-get -y update
$ sudo apt-get -y upgrade
$ sudo apt-get -y install build-essential
$ sudo apt-get -y install libsqlite3-dev
$ sudo apt-get -y install libreadline6-dev
$ sudo apt-get -y install libgdbm-dev
$ sudo apt-get -y install zlib1g-dev
$ sudo apt-get -y install libbz2-dev
$ sudo apt-get -y install sqlite3
$ sudo apt-get -y install tk-dev
$ sudo apt-get -y install zip
$ sudo apt-get -y install libssl-dev
```

如 LIST 1.1 所示，我们在执行命令时添加了 -y 选项。这样一来，在安装过程中被询问 Yes 或 No 时，计算机会自动帮我们选择 Yes。如果各位想亲自逐个确认，可以将 -y 选项删去。

build-essential 包可以批量安装 Python 在 Ubuntu 上进行构建时所需的全部工具（gcc、make 等）。Python 本身在涉及某些包和模块时也必须有这些基本工具才能进行安装，因此建议各位先装好它们。

接下来安装 Python 相关的包（LIST 1.2）。

LIST 1.2　安装 Python 相关的包

```
# 安装 python-dev
$ sudo apt-get -y install python-dev

# 安装 pip
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python get-pip.py
```

pip 是管理 Python 第三方库的工具。虽然它也能通过 apt 命令进行安装，但那样安装的版本较低，因此我们使用 `get-pip.py` 来安装最新版。

至此 Python 相关的包已经安装完毕。最后我们来查看一下 Ubuntu 默认自带的 Python 的版本。

LIST 1.3 查看 Python 的版本

```
$ python -V
Python 2.7.6
```

输入 LIST 1.3 中的命令，我们便能够查看到当前安装的 Python 版本。Ubuntu 默认安装的是 Python 2.7.6。

专栏 ensurepip

Python 3.4 具有 ensurepip 模块，可在安装 Python 的同时捆绑安装 pip。这个模块是在 Python 2.7.9 之后加入的。支持 ensurepip 的 Python 可以直接使用 pip，不需要执行 `get-pip.py` 来进行安装。另外，执行 `python -m ensurepip -U` 可以将 pip 更新到当前最新版本。

ensurepip
  `https://docs.python.org/2.7/library/ensurepip.html`
PEP 477: Backport ensurepip (PEP 453) to Python 2.7
  `https://www.python.org/dev/peps/pep-0477`

1.1.2 安装第三方包

用 `pip install` 命令可以安装第三方开发的包。

第三方包注册在 PyPI 上。这是一个用来共享 Python 包的版本库 (Repository)，任何人都可以将 Python 包上传到上面，同时也可以从上面自由下载想用的包。用 Python 开发软件时，常常要从 PyPI 安装所需的包。

NOTE

PyPI 的读音是 /'paɪˌpi: ai/。

PyPI 的构造类似 Perl 中的 CPAN[①]、Ruby 中的 RubyGems[②]、PHP 中的 PEAR[③] 等。

从 PyPI 安装包时需要用到 pip 命令。

NOTE

有关 pip 的详细内容我们将在第 3 章中详细了解。

---

① `http://www.cpan.org/`
② `http://rubygems.org/`
③ `http://pear.php.net/`

这里简单了解一下 pip 的使用方法。首先我们来查看当前安装的 pip 的版本（LIST 1.4）。

LIST 1.4　查看 pip 的版本

```
$ pip --version
pip 1.5.6 from /usr/local/lib/python2.7/dist-packages/ (python 2.7)
```

可以看到当前版本为 1.5.6（2014 年 12 月 9 日时）。

NOTE

上面例子中的 pip 安装在 dist-packages 目录下，只有通过 debian 和 ubuntu 包安装 Python 时才会安装到这个特有的位置。如果是通过 Linux 发行版、OS X、源码安装，则会安装在 site-packages 目录下。

通过 pip 安装第三方包的方法如下。首先我们来安装常用的 virtualenv 包（LIST 1.5）。

LIST 1.5　通过 pip 安装包

```
$ sudo pip install virtualenv
-（中间省略）-
Successfully installed virtualenv
Cleaning up...
```

此后，包的安装基本都要用 pip 命令来完成。另外，上面例子中的 virtualenv 包安装在 /usr/local/lib/python2.7 目录下。

NOTE

pip 在安装一些需要构建的包时，会用 gcc 等编译器进行构建。在本书所用的 Ubuntu 下，我们已经通过 build-essential 安装了 gcc。Windows 下需要用 Microsoft Visual C++ Compiler for Python 2.7 或者 MinGW 等进行安装。这些都是免费的。

1.1.3　virtualenv 的使用方法

前面安装的 virtualenv 是用来搭建虚拟 Python 执行环境的。我们将其称为 virtualenv 环境。使用这个环境时，包不会安装到 /usr/local/lib/python2.7 下，而是安装到虚拟 virtualenv 环境中。

由于我们在前面已经完成了安装，所以这里可以直接查看版本（LIST 1.6）。

LIST 1.6　查看 virtualenv 的版本

```
$ virtualenv --version
1.11.6
```

NOTE

有关 virtualenv 的详细内容我们将在第 3 章中详细了解。

我们在未使用 virtualenv 的状态下查看当前版本（LIST 1.7）。

LIST 1.7　用 pip freeze 查看当前安装版本

```
$ pip freeze
PAM==0.4.2
Twisted-Core==13.2.0
apt-xapian-index==0.45
argparse==1.2.1
configobj==4.7.2
:
:
```

输入 `pip freeze` 命令后，我们便能看到所有安装在 /usr/local/lib/python2.7 目录下的包。

接下来新建 virtualenv 环境。我们在主目录（HOME 目录）下创建工作目录，然后在这个工作目录下搭建 virtualenv 环境（LIST 1.8）。

LIST 1.8　搭建 virtualenv 环境

```
$ mkdir ~/work
$ cd ~/work
$ virtualenv venv
```

执行 LIST 1.8 中的命令后，work 目录下会自动生成 venv 目录。这就是 virtualenv 环境的目录。

接下来我们启动 virtualenv 环境。

LIST 1.9　启动 virtualenv 环境

```
$ source venv/bin/activate
(venv)$
```

如 LIST 1.9 所示，通过 `source` 命令执行 `activate`，启动 virtualenv 环境。如果终端的开头显示了“`(venv)`”（图 1.1），就证明 virtualenv 环境已经启动。



图 1.1　activate 后的 virtualenv 的状态

我们在启动了 virtualenv 的状态下再次执行 `pip freeze`，查看当前已安装的包（LIST 1.10）。

LIST 1.10　在虚拟环境下查看版本

```
(venv) $ pip freeze
argparse==1.2.1
wsgiref==0.1.2
```

可以看到，在 virtualenv 环境刚刚搭建完成时，这个 Python 执行环境中没有添加安装任何包（argparse、wsgiref 是与 Python 本体捆绑的标准库）。

想关闭 virtualenv 环境时，输入 LIST 1.11 中的命令即可。

LIST 1.11　关闭 virtualenv 环境

```
(venv) $ deactivate
$
```

如果我们不再需要某个 virtualenv 环境（本例中是 venv 目录），则可以直接用 `rm -R venv` 等命令将其连同所在目录一起删除。

1.1.4　多版本 Python 的使用

目前，Python 3.X 系列和 2.X 系列的开发是并行的，因此我们会遇到不同项目使用不同版本的 Python 的情况。本小节中，我们将学习如何在一个客 OS 环境下准备多个版本的 Python，以满足不同应用的需要。

Ubuntu 14.04 默认安装了 Python 2.7.6 和 Python 3.4.0。

LIST 1.12　查看 Python 的版本

```
$ python3 -V
Python 3.4.0

$ python -V
Python 2.7.6
```

现在我们将 Python 2.7 系列的最新版本 Python 2.7.9[①] 安装到客 OS（Ubuntu 14.04）上。

Ubuntu 14.04 上安装 Python 的方法有下列几种。

- 通过 Ubuntu 官方 deb 包安装
- 通过源文件构建安装
- 通过 PPA（Personal Package Archives，个人软件包档案）以 deb 包的形式安装

通过 deb 包的安装十分简便，但包中封装的不一定是最新版的 Python。某些版本的 Python

① `http://www.python.org/downloads/release/python-279/`

含有重要更新，这时候就需要我们通过源文件进行构建了。鉴于这类情况，我们将在这里介绍“通过源文件构建”这一安装方式（LIST 1.13）。

LIST 1.13　通过源文件构建并安装

```
$ wget https://www.python.org/ftp/python/2.7.9/Python-2.7.9.tgz
$ tar xvzf Python-2.7.9.tgz
$ cd Python-2.7.9
$ LDFLAGS="-L/usr/lib/x86_64-linux-gnu" ./configure --prefix=/opt/python2.7.9
$ make
$ sudo make install
```

该 Python 将被安装到 /opt/python2.7.9/bin 下，我们可以通过文件名中的版本号找到它。现在执行 Python 命令查看是否安装成功（LIST 1.14）。

LIST 1.14　查看 Python 2.7.9

```
$ /opt/python2.7.9/bin/python -V
Python 2.7.9
```

到这里，Python 的安装就结束了。接下来我们看看如何在开发应用时区分使用各个版本的 Python。

NOTE

像 Python 2.6 这种官方 apt 版本库早已不支持的旧版本，我们可以通过 PPA，即分享个人软件包的 apt 版本库进行安装（LIST 1.15）。但要注意，PPA 中的软件包全都是个人上传的，一切使用后果要由自己负责。

LIST 1.15　通过 PPA 以 deb 包的形式安装

```
$ sudo add-apt-repository ppa:fkrull/deadsnakes
$ sudo apt-get -y update
$ sudo apt-get -y install python2.6
$ python2.6 -V
Python 2.6.9
```

NOTE

Python 的部分模块依赖于一些必须通过 apt 命令进行安装的包。对于这类 Python 模块，如果不事先用 apt 命令安装好包，则会发生 ImportError。

这种时候我们需要先用 apt 命令安装所需的包，然后再通过源文件进行构建。因此如果各位遇到 Python 模块无法使用的情况，可以考虑重新通过源文件构建一遍。有些 Python 的标准

模块需要先安装 deb 包才能使用，具体内容如下所示。

Python 模块	deb 包
zlib	zlib1g-dev
sqlite3	libsqlite3-dev
readline	libreadline6-dev
gdbm	libgdbm-dev
bz2	libbz2-dev
Tkinter	tk-dev

◉ 借助 virtualenv 分别使用不同版本的 Python

当 OS 中安装了多个 Python 时，我们可以通过完整路径指定 Python 命令，或者使用带版本号的 Python 命令（如 `python2.7` 和 `python3.4`）来区分使用各版本。

不过，如果在搭建 virtualenv 环境时指定了该环境下的默认 Python，那么在启动该环境时，我们就不需要顾虑版本问题了。

LIST 1.16　指定 virtualenv 下执行的 Python

```
$ virtualenv --python=/opt/python2.7.9/bin/python venv2
$ source venv2/bin/activate
(venv2)$
Python 2.7.9
```

根据 LIST 1.16 进行指定之后，以 /opt/python2.7.9/bin/python 为基础的 virtualenv 环境——venv2 就搭建完成了。

1.2　安装 Mercurial

版本控制系统在如今的开发过程中已经十分普及，Python 自然也不例外。人们使用版本控制系统通常是为了多人一起管理源码，不过，将它引入到个人开发中也能带来不少好处。比如在开发过程中应用突然不工作了，我们就可以将其回溯成能正常工作的版本。另外，对版本进行管理之后，一旦哪里发现了问题，我们可以立刻沿时间线查找之前的版本，快速找出问题发生的时间点。

版本控制系统大致可分为两类，一类是 CVS 和 Subversion 这种单一版本库的版本控制系统，另一种则是 Mercurial 和 Git 这种分布式版本控制系统（Distributed Version Control System，DVCS），如今越来越多人倾向于使用后者。Mercurial 这种分布式版本控制系统最大的优势在于

各个版本库独立，因此在创建分支、提交、取消更改时不会对周围人造成影响。而且分布式的版本库可以各自看作是彼此的备份。

接下来我们来了解一下 Mercurial。

1.2.1 Mercurial 概述

Mercurial 是 Linux 内核的开发者 Matt Mackall 于 2005 年开始开发的分布式版本控制系统。Linux 内核之父 Linus Torvalds 也于同期开始开发 Git。如今这二者已经成为分布式版本控制系统的代表，被人们广泛使用。

由于 Mercurial 本身就是由 Python 编写而成的，因此我们可以轻松地通过 pip 进行安装，也可以将其视为 Python 程序进行自定义设置。

对熟悉 Python 的用户而言易于上手，与使用 Python 进行开发的团队亲和力高，这些都是 Mercurial 的优势所在。

此外，Mercurial 的魅力还在于内置了基于 Web 的管理工具，支持 Bitbucket① 等著名源码托管服务，TortoiseHg② 等 GUI 客户端丰富，等等。

不仅如此，Mercurial 的命令体系与广为人知的 Subversion 很类似，对于长期使用 Subversion 的用户而言，要比 Git 更容易上手。

1.2.2 安装 Mercurial

单看前面的说明很难对分布式版本控制系统的优势有一个明确理解，所以现在我们通过实际的安装和操作来体会一下。另外，高级的 Mercurial 运用技巧我们将在第 6 章中进行学习，熟悉 Mercurial 的读者可以直接跳过本部分。

Mercurial 与其他程序包一样具有多种安装方法，用户可以通过 `apt-get`、`pip`、源文件构建等不同途径进行安装。这里我们为保证 Mercurial 尽量是最新版，决定选用 pip 进行安装（LIST 1.17）。

LIST 1.17 安装 Mercurial

```
$ sudo pip install mercurial
```

Mercurial 安装完成后我们就可以使用 hg 命令了。现在输入 `hg --version` 来查看版本（LIST 1.18）。

---

① http://bitbucket.org/

② http://tortoisehg.bitbucket.org/

LIST 1.18　查看 Mercurial 的安装情况

```
$ hg --version
Mercurial Distributed SCM (version 3.1.2)
(see http://mercurial.selenic.com for more information)

Copyright (C) 2005-2014 Matt Mackall and others
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1.2.3　创建版本库

在创建版本库之前，我们先准备好 Mercurial 的环境设置文件。在主目录下创建名为 .hgrc 的文件，然后进行如下描述。其中括号内的部分称为节（Section），比如 [ui] 节就是用来设置 username 属性的。

username 中的账户信息会在用户向版本库进行提交时记录在日志中。各位请将自己的账户名和邮箱地址写在这里（LIST 1.19）。

LIST 1.19　设置 .hgrc 的用户名与邮箱地址

```
[ui]
username=bpbook <bpbook@beproud.jp>

[extensions]
color=
pager=

[pager]
pager=LESS='FSRX' less
```

[extensions] 节用来激活 Mercurial 附属的扩展工具。只要将扩展工具名写入 [extensions] 节的项目中，我们就可以使用该工具了。此外，某些扩展工具不仅要在 [extensions] 中设置激活，还需要在 [pager] 节中设置 pager 属性。

完成环境设置文件之后就该创建版本库了。如 LIST 1.20 所示，我们为版本库创建一个目录并移动至该目录下，执行 `hg init` 命令创建版本库。

LIST 1.20　hg init（初始化版本库）

```
$ mkdir ~/hgtest
$ cd ~/hgtest
$ hg init
```

1.2.4　文件操作

创建好版本库之后让我们给它实际添加文件。现在创建一个测试用的文件，查看当前版本库的状态。

查看版本库状态用 `hg status` 命令（LIST 1.21）。如果觉得每次输入 `status` 太麻烦，可以将这个命令缩写为 `hg st`。hg 命令为用户准备了部分缩写形式，感兴趣的读者可以参考帮助或其他资料。

LIST 1.21　hg status（查看状态）

```
$ touch test.txt
$ hg status

? test.txt
```

文件名左侧显示了该文件在版本库内的状态。各位可以看到 text.txt 文件的左侧显示了状态“?”，这代表该文件现在并不在版本管理的范围内。因此我们执行 `hg add` 命令将其添加为版本管理的对象，如 LIST 1.22 所示。

LIST 1.22　hg add（添加文件）

```
$ hg add test.txt
$ hg status

A test.txt
```

执行 `hg add` 之后我们再进行一次查看，会发现之前的“?”已经变成了“A”。这代表该文件是新添加到版本管理里的。

要想让添加文件反映到版本库，我们需要进行提交。提交时请按照 LIST 1.23 所示，输入 `hg commit` 命令。如果不执行 `hg commit`，那么我们创建的文件就只能停留在当前工作的机器里，不会反映到版本库中。

LIST 1.23　hg commit（提交）

```
$ hg commit

test commit

HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: bpbook <bpbook@beproud.jp>
```

```
HG: branch 'default'
HG: added test.txt
```

执行 `hg commit` 后编辑器就会启动，用来记录提交时的相关信息与注释。我们在这里输入 `test commit`。

注释输入栏下方显示着 `branch 'default'`，这表示 default 分支（版本库创建时就存在的分支）为我们当前进行修正 / 添加操作的对象。

NOTE

提交时打开的编辑器是可以更改的。更改时需要如下例所示，在 .hgrc 的 [ui] 节里添加 editor 属性。

```
[ui]
editor = vim
```

这表示我们要使用 vim 编辑器。默认启动的编辑器取决于环境变量 EDITOR，环境变量可以用下述命令查看。

```
$ echo $EDITOR
vim
```

提交文件之后用 `hg status` 查看，结果应该如 LIST 1.24 所示，什么都没有。

LIST 1.24　提交后的查看

```
$ hg status
```

接下来我们看看如何在 Mercurial 中操作编辑过的文件。

先用编辑器打开 test.txt 进行修改，然后保存文件。在当前状态下执行 `hg status` 后，会发现该文件的状态变成了 `M`。状态 M 表示该文件在最后一次提交之后又进行了“变更”（Modify）。使用 `hg diff` 命令可以查看变更前的状态（已提交的文件）与当前状态的差别（LIST 1.25）。

LIST 1.25　hg diff（查看差别）

```
$ hg status
M test.txt

$ hg diff

diff -r 74471564b074 test.txt
--- a/test.txt      Mon Oct 31 18:07:15 2014 +0900
+++ b/test.txt      Mon Oct 31 18:10:45 2014 +0900
```

```
@@ -0,0 +1,1 @@
+modify this file
```

在尚未提交的状态下，我们可以使用 hg revert 命令撤销编辑（LIST 1.26）。

LIST 1.26 hg revert（撤销编辑）

```
$ hg revert test.txt
```

NOTE

hg revent 可以撤销尚未提交的内容，但对已提交的内容无效。这种情况需要使用 hg commit --amend 命令对已提交的变更集进行修改或撤销。

Mercurial 的基本操作方法就解说到这里，但下列几项我们并未提及。

- 分支的操作
- 远程版本库的使用
- 以团队开发为前提的 Mercurial 的使用方法

上述几点我们将在第 4 章和第 6 章中详细了解。

1.3 编辑器与辅助开发工具

本节将介绍编写 Python 代码时可用的编辑器，以及一些有助于开发的小贴士和 Python 模块。

1.3.1 编辑器

编写 Python 代码自然少不了编辑器。当今世上的编辑器种类繁多，但编写 Python 代码的编辑器必须具备下列几项功能。

- 语法高亮

  将语法设置高亮是一项十分重要的功能。用特殊字色或字体表示关键词可以让编程人员快速发现拼写类错误。这样一来，我们便能够在编写代码的过程中注意并修正此类错误。
- 智能缩进

  自动根据语法添加缩进的功能。特别是对于Python这类缩进具有重要意义的语言，编辑

器适时地自动缩进要比手动插入空格时的编程效率高得多。

- 执行 DEBUG

  在编辑器上执行DEBUG的功能。比如在编程过程中遇到某些问题时，如果能直接使用编辑器进行DEBUG，那么将能更快地找出原因，提高解决问题的效率。

- 可以使用静态解析类插件

  能在编辑器上执行静态解析类插件的功能。如果能直接在编辑器内执行语法高亮无法查出的键入错误、语法错误、编码格式检查等，将能省去每次开关编辑器的麻烦，大幅削减修改时消耗的时间。

这里将按照上述 4 点，给各位介绍 3 个编辑器（Vim、Emacs、PyCharm）的使用方法和功能。

各位可以以此为参考来寻找自己用着顺手的编辑器。

◉ Vim

Vim 是一款功能强大的文本编辑器，默认搭载在许多 OS 上，普及度很高。

Vim 可以通过内置的 Vim script 脚本语言进行功能扩展，因此存在许多插件。

- 语法高亮

  默认捆绑Python的语法高亮设置文件（python.vim）。

- 智能缩进

  有自动缩进的设置选项。缩进时插入的空格数也可以调节。

- 执行 DEBUG

  Vim可以在编辑器上调出外部命令，因此可以执行`pdb`（后述）。比如执行`:!/path/to/virtualenv/python -m pdb %`之后，编辑器便会对编辑中的脚本启动调试工具。

- 可以使用静态解析类插件

  vim-flake8插件可以检查当前打开的文件并发出问题警告。

Vim 的设置在主目录“`.vimrc`”文件中进行，我们写在该文件中的设置将反映在 Vim 上。现在我们来设置语法高亮和自动缩进（LIST 1.27）。

LIST 1.27　用于 Python 的设置

```
" 语法高亮的设置
syntax on
" 自动缩进的设置
filetype plugin indent on
```

另外，设置文件本身也可以根据对象文件类型进行分割。下面，我们专门为 Python 脚本准

备一个设置文件（LIST 1.28）。

LIST 1.28　准备 Python 专用的设置文件

```
$ mkdir ~/.vim
$ mkdir ~/.vim/ftplugin
$ touch ~/.vim/ftplugin/python.vim
```

我们在这个 python.vim 中进行针对 Python 的设置，以方便编写 Python 代码。

LIST 1.29　将 Tab 改为 4 个空格

```
" 将“Tab”替换为“空格”
setl expandtab
" 将“Tab”的“缩进幅度”改为 4
setl tabstop=4
" 自动缩进时的“缩进幅度”改为 4
setl shiftwidth=4
" 按下键盘“Tab”键时插入的空格数
" 这里设置为 0 就可以插入“tabstop”中设置的空格数了
setl softtabstop=0
" 保存时删除行尾的空格
autocmd BufWritePre * :%s/\s\+$//ge
"80 个字符换行
setlocal textwidth=80
```

像 LIST 1.29 这样设置是为了迎合 Python 社区推荐的 PEP 8① 编码格式（后述）。

vim-flake8 是 Vim 的插件，需要进行安装。neobundle.vim② 是一个专门用来管理 Vim 插件的插件，用它可以轻松安装其他 Vim 插件。

安装 neobundle.vim 后，我们在“.vimrc”里这样设置 vim-flake8。

```
NeoBundle 'nvie/vim-flake8'
```

现在只要重启 Vim 就会开始安装插件。安装结束后，用 Vim 打开 Python 脚本，按下 Ctrl + F7 就可以看到 flake8 的警告信息了。

NOTE

python.org 官方 Wiki 上也记载着 Vim 的一些设置资料，各位不妨去看一看。

・Vi Improved③

① https://www.python.org/dev/peps/pep-0008
② https://github.com/Shougo/neobundle.vim
③ https://wiki.python.org/moin/Vim

此外，在 Vim 官方网站和许多民间社区中都能找到大量关于 Vim 的信息。

- Vim online①
- vim-jp②

◉ Emacs

Emacs 是 Unix 系列 OS 长期沿用的编辑器。该编辑器可以用 Emacs Lisp 语言进行扩展，因此拥有大量插件。如今的 Emacs 已经捆绑了 python.el，安装 Emacs 之后可以直接编写 Python 代码，不必特意去设置。

人们常说 Emacs 是一个环境，不过我们在这里主要是看它作为 Python 编辑器的特征。

- 语法高亮
  内置python.el。当`global-font-lock-mode`不为`nil`时（默认）进行语法高亮。
- 智能缩进
  python.el会按照PEP 8的要求缩进到指定位置。
- 执行 DEBUG
  Emacs上可以执行pdb。
  先键入`M-x pdb`，然后直接在minibuffer里运行类似`/path/to/virtualenv/python -m pdb /path/to/app.py`的pdb命令行。此时会启动pdb专用的缓冲，我们可以在里面查看源码或进行调试。
- 可以使用静态解析类插件
  `flymake-python-pyflakes`可以在编码过程中高亮警告。

安装完成之后我们在设置文件 $HOME/.emacs.d/init.el 中设置 Tab 和空格的执行动作。另外，Emacs 可以让空格和 Tab 文字可视化。在 Python 中，缩进是语法的一部分，所以建议各位先将这些设置好（LIST 1.30）。

LIST 1.30　空格和 Tab 的设置

```
(require 'whitespace)
(setq whitespace-style '(face tabs tab-mark spaces lines-tail empty))
(global-whitespace-mode 1)
(setq-default tab-width 4 indent-tabs-mode nil)
(setq indent-tabs-mode nil)
```

flymake-python-pyflakes 等插件并没有捆绑在 Emacs 中，所以我们需要通过 Emacs 的版本

① `http://www.vim.org/`

② `http://vim-jp.org/`

控制系统 elpa 进行安装。首先在 $HOME/.emacs.d/init.el 文件中进行如 LIST 1.31 所示的设置。

LIST 1.31 elpa 的程序包版本库设置

```
(require 'package)
(add-to-list 'package-archives
         '("melpa" . "http://melpa.milkbox.net/packages/"))
(add-to-list 'package-archives
         '("marmalade" . "http://marmalade-repo.org/packages/"))
(package-initialize)
```

通过 elpa 安装好 flymake 和 flymake-python-pyflakes 之后，我们再进行 LIST 1.32 所示的设置。

LIST 1.32 flymake 的设置

```
(require 'flymake)
(require 'flymake-python-pyflakes)
(add-hook 'python-mode-hook 'flymake-python-pyflakes-load)
(setq flymake-python-pyflakes-executable "flake8")
```

设置好 flymake，我们编写源码时就能在 Emacs 上看到 flake8 的警告了。

> **NOTE**
>
> python.org 上有设置 Emacs 的相关资料。
>
> · Emacs Editor①
>
> 另外，我们在其官方网站和许多 Wiki 上也能找到大量 Emacs 的信息。
>
> · GNU Emacs② · EmacsWiki③ / · Python Programming In Emacs④

◉ PyCharm

对长期以来使用 Eclipse 等 IDE（Integrated Development Environment，集成开发环境）进行开发的人来说，IDE 用起来要比编辑器顺手得多。在当今众多的 IDE 中，普及率较高的当属 PyCharm⑤ 了。

PyCharm 是一款 2010 年左右问世的 IDE，为 Windows、OS X、Linux 等 OS 提供了相应的安装程序，同时还支持 Python 的 Web 框架 Django。PyCharm 有多种许可证形态，比如付费的

---

① https://wiki.python.org/moin/EmacsEditor
② http://www.gnu.org/software/emacs/emacs.html
③ http://www.emacswiki.org/
④ http://www.emacswiki.org/emacs/PythonProgrammingInEmacs
⑤ http://www.jetbrains.com/pycharm/

Pro 版和免费的 Community 版等。其中付费的 Pro 版还提供 30 天的免费使用期。

与 Vim 和 Emacs 相比，PyCharm 自带了许多 Python 开发的辅助功能，在安装完之后可以立刻使用。相对地，自行开发插件则需要应用 Java 的知识，所以并不算容易。从这种角度来讲，这款工具可能并不适合极客们使用。

如果各位实在无法适应 Vim 或 Emacs，那么建议先尝试 PyCharm 的 Community 版。它作为编写 Python 时使用的 IDE，具有以下特征。

- 语法高亮

  标准功能。PyCharm会根据当前项目使用的Python版本切换语法及内置函数等文字的高亮效果。

  PyCharm标准支持Python、HTML、CSS、JavaScript、XML、SQL以及CoffeeScript、Angular JS、LESS、SASS、SCSS等格式的高亮显示。Pro版还支持Django、Mako、Jinja2、Web2py、Chameleon模板的记法。

- 智能缩进

  标准功能。PyCharm会按照PEP 8的要求缩进到指定位置。

  与语法高亮相同，PyCharm也支持Python以外的其他格式。另外，用户还可以对各个项目的各个语言进行详细设置，比如规定插入Hard Tab还是空格等。

- 执行 DEBUG

  标准功能。IDE中可以设置断点、逐句执行、显示执行中的变量值。Pro版可以用手边终端的IDE调试远程服务器上的进程。

- 静态解析类功能

  标准功能。在编写代码的过程中，PyCharm会高亮显示对未使用的变量、未定义的名称、已过期函数的警告等。另外，PyCharm在执行解析功能时会从多角度分析Python代码是否存在风险。

导入 PyCharm 之后，我们就可以不再为搭建环境劳神费心了。它为用户标配了大量功能，大部分人都可以在默认设置的状态下轻松使用。不过相对地，一旦习惯了这些懒办法，当极客的乐趣自然也会大打折扣。

PyCharm 也并不适用于所有情况，比如编辑单个文件就很麻烦。这种情况需要先创建一个空项目，把待编辑的文件加入项目之后才能打开编辑。另外，PyCharm 的使用环境建议内存为 2 GB 以上，显示器为 SXGA 以上。

专栏 PyCharm 适合用来干什么

PyCharm3 及之前的版本以提供面向 Web 开发者的辅助功能为主。从 2014 年 11 月发布的 PyCharm4 开始，该系列 IDE 开始内置 IPython notebook 等功能，预计今后会逐步增添科学计算相关的辅助功能。

1.3.2 开发辅助工具

本部分将向各位介绍一些 Python 开发过程中应当了解的模式及 Python 包。

◉ 交互模式

单独执行 `python` 命令时，该命令会以交互模式执行。交互模式指通过对话方式输入并执行 Python 代码的模式。现在请直接输入 `python` 命令（LIST 1.33）。

LIST 1.33 启动交互模式

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

随后系统会进入等待输入的状态。我们可以在这个状态下直接编写 Python 代码。代码执行后的内容会直接显示在屏幕上（LIST 1.34）。

LIST 1.34 Python 代码执行示例

```
>>> import sys
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload', '/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages']
>>>
```

我们导入了标准模块 sys 并显示了 Python 包的搜索路径列表。

最后一次执行的结果存储在变量“_”中，执行它就可以返回相同结果。交互模式让我们能随手轻松地查看代码，方便进行 DEBUG 等操作。如果想结束交互模式，可以按 Ctrl + d 或输入 `exit()`。

NOTE

交互模式在启动时会读取环境变量 PYTHONSTARTUP 中设置的文件。如果各位在启动交互模式时想顺便导入些什么，可以在这里进行设置。

另外，IPython 模块可以将 Python 的交互模式用作调试器。该模块可通过 pip 安装。

```
$ sudo pip install ipython
```

导入 IPython 后，交互模式会实现下述功能。

- TAB 键补全代码
- 使用通常的 Shell 命令
- 与 pdb（调试器）联动

◉ flake8（编码格式 / 语法检测）

为更好地规范和优化 Python，人们以社区为中心给 Python 制定了 PEP（Python Enhancement Proposals，Python 增强建议书）[①] 指导规范。

其中 PEP 8 主要规范 Python 的编码格式。这里即将介绍的 flake8 模块就可以检查我们的代码是否符合该编码格式。如 LIST 1.35 所示，flake8 可通过 pip 安装。

LIST 1.35　安装 flake8

```
$ sudo pip install flake8
```

flake8 不但能检查编码格式，还可以帮助我们找出语法错误以及一些已导入但未被使用的模块并发出警告。

使用方法很简单，只要像 LIST 1.36 一样指定文件并执行 flake8 命令即可。

LIST 1.36　flake8 执行示例

```
$ flake8 sample.py
sample.py:1:12: E401 multiple imports on one line
sample.py:1:17: E703 statement ends with a semicolon
sample.py:3:1: E302 expected 2 blank lines, found 1
sample.py:4:5: E265 block comment should start with '# '
sample.py:4:80: E501 line too long (83 > 79 characters)
sample.py:5:15: E225 missing whitespace around operator
```

① https://www.python.org/dev/peps

NOTE

由于我们平时使用的是 flake8，所以这里只对它进行了介绍。该类常用的程序包还有 PyLint 等，各位可以根据自己的开发风格选用合适的模块。

◉ pdb（调试器）

最后我们来了解一下 Python 的调试器。如果各位曾经用过 C 语言的 gdb 等调试器，或者对 IDE 等上面附属的调试器有所接触，那么一定不会对插入断点、逐句执行等功能感到陌生。Python 的调试器也具备这些功能。

pdb 是 Python 的标准模块，不必另外安装。其最简单的用法就是在我们希望程序停止的位置插入如 LIST 1.37 所示的代码。

LIST 1.37　pdb 的插入代码

```
def add(x, y):
    return x + y


x = 0
import pdb; pdb.set_trace()
x = add(1, 2)
```

执行这个插入了 LIST 1.37 中的代码的 Python 脚本时，脚本会停在上述插入代码的位置，然后启动对话型界面。

LIST 1.38　pdb 执行示例

```
$ python pdbtest.py
> pdbtest.py(7)<module>()
-> x = add(1, 2)
(Pdb)
```

本节我们对一些辅助开发的工具、Python 程序包等作了了解。由于涉及范围较广，我们只简单学习了部分使用方法。有关这些工具、程序库的详细信息请各位自行查阅相关文档。

1.4　小结

本章对下列话题进行了介绍，意在指导各位完成 Python 语言独立开发的事前准备。

- 安装 Python

- 安装 Mercurial
- 编辑器和 DEBUG 工具

为保证开发前的准备工作，本章介绍了许多工具的安装方法。这里介绍的很多内容在 Ubuntu 以外的 OS 上同样适用。此外，各位还可以编写自动完成安装的 Shell 脚本，或者省略并简化本书介绍的步骤等，总之能做的事情还有很多。各位请根据自己的需要搭建适合自己的环境。

NOTE

进入第 2 章之前，我们先来看一下 The Zen of Python。The Zen of Python 是 Python 开发者之一 Tim Peters 撰写的文章，通过 19 个分项简明扼要地表达了 Python 的特点。The Zen of Python 现已作为 PEP 20[①] 公开发表，在交互模式下输入 import this 命令也可以看到这篇文章。我们强烈建议各位去读一读。

```
>>> import this
```

另外，@atsuoishimoto 在博客上对 The Zen of Python 进行了详细讲解，各位不妨参考一下。

・The Zen of Python 解题 - 前篇[②]（日语）

・The Zen of Python 解题 - 后篇[③]（日语）

① https://www.python.org/dev/peps/pep-0020

② http://www.gembook.org/2010-09-20.html

③ http://www.gembook.org/2010-09-26.html

第 2 章　开发 Web 应用

本章中，我们先来了解 Web 应用的概念，然后一起实际开发一个简单的留言板应用，借此来了解 Web 应用开发的基本流程。至于留言板应用，它类似于观光景点的留言板，可以让访问网站的人在上面添加留言，说白了就是一个简单的网络留言板。正文中会涉及 HTML/CSS 和 Python 代码，但本书将省去对这些语法的说明。另外，本章的开发环境为 VirtualBox 虚拟机上的 Ubuntu 和 Python 2.7，同时还会用到 virtualenv。

2.1　了解 Web 应用

要想开发 Web 应用，首先要知道 Web 应用是什么，它是怎样工作的。所以我们先来了解一下什么是 Web 应用。

2.1.1　Web 应用是什么

顾名思义，Web 应用就是可以通过网络使用的应用程序。比如 Google 的搜索服务、Gmail、Wikipedia、各种博客服务、Twitter 等迷你博客、GREE 和手机游戏、Facebook 等社交网站以及上面的社交游戏等，这些都是 Web 应用。人们使用 Web 应用时需要通过 Web 浏览器访问（连接）相应服务。

那么，同样需要通过 Web 浏览器阅览的 Web 站点又如何呢？ Web 站点只是单纯地显示页面，它们能称作 Web 应用吗？

答案是不一定。某些 Web 站点或许可以称为 Web 应用。因为虽然有些东西在阅览者眼中像 Web 站点，但它实际上却是由 CMS（Content Management System，内容管理系统）等 Web 应用构成的。Web 浏览器会从访问对象的计算机（服务器）中下载 HTML、CSS、图片等各种内容，然后再把这些内容显示在我们的屏幕上。如果负责发送内容的服务器只是返回一些早已准备好的静态内容，那么这个 Web 站点就不能称作 Web 应用。只有能够动态生成并返回内容的系统（比如通过 Web 浏览器接收用户输入的数据，再根据这些数据生成内容）才能称作 Web 应用。

NOTE

CMS 是负责管理和发送文章、图片等内容的系统。Wiki 和博客系统也都属于 CMS。

2.1.2　Web 应用与桌面应用的区别

要通过 Web 浏览器连接服务器使用的应用叫 Web 应用；相对地，要将软件安装在计算机上才能使用的应用叫桌面应用。二者的区别如下。

比较项目	Web 应用	桌面应用
服务器	需要	可有可无
网络连接	需要	部分应用需要
使用 OS 具备的功能	不可能	可能
安装	不需要	需要
升级	不需要使用者专门注意	需要使用者亲自动手
跨平台使用	有 Web 浏览器就能用	需要各平台分别作处理
移动端支持	与 PC 端基本相同	需要各平台分别作处理
运行速度	慢	快

从这张表上看，二者各有所长。但如果要开发一个让使用者通过网络互相交流信息的应用，那么由于桌面应用也同样需要用到服务器，所以使用 Web 应用开发起来会简单很多。不过，这个表中的内容也不是一成不变的。

比如 Web 应用本来离不开网络，但如今已有一部分应用可以离线运行。另外，某些桌面应用也开始具备自动升级新版本的功能（比如 Google Chrome），这让使用者不必再费心手动升级。可以说，Web 应用和桌面应用之间的距离正在逐渐缩小。

2.1.3　Web 应用的机制

接下来我们来了解一下 Web 应用的机制。

从我们在 Web 浏览器中输入 URL 到显示出页面，其间要进行下述处理。

① 用户在 Web 浏览器中输入 URL

② 向 DNS 服务器询问该 URL 中的域名，获取 IP 地址

③ Web 浏览器连接该 IP 地址的 Web 服务器，开始 HTTP 通信

④ Web 服务器根据 HTTP 发送来的信息运行 Web 应用，获取相应内容

⑤ Web 服务器响应，返回执行应用后得到的 HTML、CSS、JavaScript、图片文件等内容

⑥ Web 浏览器将收到的内容显示在页面中

下表是对上述处理流程中的术语进行的说明。

Web 浏览器	通过 HTTP 等协议与 URL 所示的计算机通信，将 HTML、CSS、图片等内容显示在页面中的软件
URL	Uniform Resource Locator（统一资源定位符）的简称，其中包含域名。这个字符串用来表示计算机需要访问网络上的哪些内容
域名	与 IP 地址挂钩的字符串（比如 URL 中包含的 www.beproud.jp 等字符串）
DNS 服务器	可以通过域名查询 IP 地址的服务器
IP 地址	这个数值用来在网络上识别我们想访问的计算机（以 IPv4 地址为例，IP 地址由 0 ~ 255 的数字和点组成，比如 192.168.0.1）
HTTP	Hypertext Transfer Protocol（超文本传输协议）的简称，是与被访问计算机之间的通信协议
HTML	用来描述文档（包括文字和图片等）结构的语言
CSS	描述 HTML 等语言描述的文档该如何显示的语言、机制
JavaScript	在 Web 浏览器上运行的程序
Web 服务器	通过 HTTP 进行通信的服务器程序 / 计算机
Web 应用	在 Web 服务器上运行的程序
CGI	Common Gateway Interface（通用网关接口）的简称，Web 应用的机制之一

上述流程可以用图 2.1 表示。

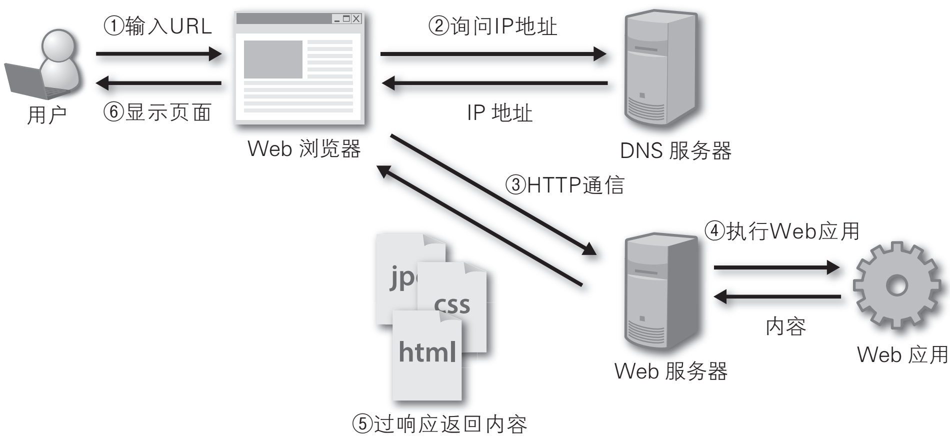


图 2.1 Web 应用的处理流程

◉ Web 应用与 CGI

CGI 是 Web 服务器运行 Web 应用的一种机制。Web 服务器执行 CGI 程序（CGI 脚本），然后将该程序的标准输出结果作为 HTTP 通信的响应返回给对方。最简单的 CGI 程序就是在控制台界面上显示字符串。

CGIHTTPServer 是 Python 标准模块中的 Web 服务器，它可以运行 CGI 程序。现在我们试着运行下面这个 CGI 程序（LIST 2.1）。

LIST 2.1　hello.py

```
#!/usr/bin/env python
print "200 OK"
print "Content-Type: text/plain"
print ""
print "Hello CGI!"
```

用 python 命令运行这个脚本，然后控制台界面中将显示如 LIST 2.2 所示的 4 行字符串。

LIST 2.2　用 python 命令运行 hello.py 的结果

```
$ python hello.py
200 OK
Content-Type: text/plain

Hello CGI!
```

用 CGIHTTPServer 运行 CGI 程序时，待运行文件必须位于 cgi-bin 目录下，所以我们要创建这个目录并将 `hello.py` 放进去。另外，必须具有运行权限才可以运行这些文件，因此还要用 `chmod` 命令赋予运行权限。配置好 CGI 程序之后，我们就可以按照下面的例子，用 python 命令的 -m 选项运行 `CGIHTTPServer` 了。

LIST 2.3　CGI 程序的配置与 CGIHTTPServer 的运行

```
$ mkdir cgi-bin
$ mv hello.py cgi-bin/
$ chmod u+x cgi-bin/hello.py
$ python -m CGIHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

执行 LIST 2.3 中的命令后，在该环境（这里是 VirtualBox 上的 Ubuntu）的 8000 端口等待请求的 Web 服务器将会启动。在终端按下 Ctrl + C 键可以关闭服务器。

要想通过本地环境（主 OS）的 Web 浏览器查看效果，我们需要先设置端口转发。设置方法与附录 B 中介绍的 SSH 端口的设置方法一样，这里我们给 VirtualBox 的端口转发添加如下设置：主、客端口均为 8000，协议为 TCP。

完成端口转发的设置后，我们只要通过 Web 浏览器访问 `http://127.0.0.1:8000/cgi-bin/hello.py`，即可看到 `Hello CGI!` 字样。

NOTE

127.0.0.1 代表的是自己的计算机的 IP 地址。设置过 VirtualBox 的端口转发之后，我们对自己的计算机（主 OS）的 8000 端口的访问会被转发到客 OS 的 8000 端口。

本例中的 CGI 程序由 Python 代码编写而成。实际上，只要 CGI 程序能够进行标准输出，用任何语言都没有问题。

◉ Web 应用与应用服务器

应用服务器指能运行 Web 应用的功能且能与 Web 服务器通信的服务器。处于启动状态的应用服务器会一直等待 Web 服务器发来请求，一旦接到请求便会运行 Web 应用并返回结果。由于待运行的程序一直放在内存里，所以它的速度通常要比 CGI 程序的速度更快。Web 服务器与应用服务器之间的通信协议通常为 HTTP 或 FCGI 等。如果一个应用服务器可以通过 HTTP 通信，那么我们就可以用 Web 浏览器访问它。

本章使用的 Flask 内置了用于开发的应用服务器（Web 服务器），因此不需要再另外准备 Web 服务器。

2.2 前置准备

接下来我们需要为开发 Web 应用做一些前置准备。这里需要使用 Python 和 virtualenv，二者的安装请参考第 1 章。

2.2.1 关于 Flask

开发留言板应用时，我们会用到 Flask[①]，因此需要事先安装。Flask 是一个用 Python 编写的 Web 应用框架，它整合了 Werkzeug（WSGI 实用工具）和 Jinja2（模板引擎）两个库。用它可以轻松完成小规模的应用程序开发。

2.2.2 安装 Flask

安装 Flask 之前，我们先用 virtualenv 命令搭建开发应用所需的虚拟环境。用 LIST 2.4 中的命令可以搭建出名为 venv 的虚拟环境。

LIST 2.4 搭建名为 venv 的虚拟环境

```
$ virtualenv venv
```

通过 `source` 命令执行 venv/bin 目录下的 activate 脚本，启动我们刚刚搭建好的虚拟环境（LIST 2.5）。执行完成后，命令提示符上会显示虚拟环境名（`venv`）。

① http://flask.pocoo.org/

LIST 2.5　启动虚拟环境 venv

```
$ source venv/bin/activate
```

> **NOTE**
>
> Windows 需要使用 venv\Scripts\activate 命令启动虚拟环境。

接下来使用 pip 命令在虚拟环境上安装最新版本的 Flask。我们需要在 venv 虚拟环境处于激活状态时执行 LIST 2.6 中的命令。

LIST 2.6　用 pip 命令安装 Flask

```
$ pip install -U Flask
```

指定 -U 选项之后，可以用新版本替换已经安装的旧版本。2014 年 12 月时，最新的 Flask 版本为 0.10.1。至此 Flask 安装完毕，前置准备结束。

2.3　Web 应用的开发流程

那么，接下来我们应该按什么顺序开发 Web 应用呢?

现在唯一确定下来的事情就是要开发一个留言板应用，除此之外毫无计划。首先我们要确定这个应用的需求，再以需求为出发点考虑如何实现页面和功能，然后开始敲代码。等编码完成后，还要测试该应用是否能够正常运行。

整个流程总结起来是下面这样的。

① 确认需求（确认要开发什么应用）
② 根据需求明确成品必备的功能
③ 根据功能明确成品必备的页面
④ 页面设计
⑤ 实现功能
⑥ 将功能植入到页面中
⑦ 确认是否能正常运行
⑧ 完成

关于上述各流程，我们将在实际开发过程中详细了解。

2.4　明确要开发什么应用

这里，我们来了解一下这个应用的需求以及必备功能和页面。

2.4.1　留言板应用的需求

首先，我们来确定留言板应用是个什么样的应用，需要实现些什么（即需求）。

如果不事先明确需求或规格，很容易在编码过程中遇到“原本想做什么来着”“这个功能有必要做吗”之类的问题，导致项目一团乱。

因此我们需要先确定这个留言板应用的需求，其具体内容如下。

① 在 Web 浏览器上显示一个包含“提交留言”表单的页面
② 可以在提交留言表单中输入名字和留言正文
③ 通过提交留言表单发送的名字和留言内容会被保存
④ 已保存的名字、留言、提交日期会显示在页面中
⑤ 整个应用由一个页面构成，页面上部为提交留言表单，下部显示已提交的内容
⑥ 提交的内容按新旧顺序由上到下排列
⑦ 可经由网络（互联网）使用本系统
⑧ 可同时在多台计算机上显示已提交的内容

2.4.2　明确必备的功能

确定好该应用的需求后，我们来思考一下该用哪些功能来满足这些需求。

这次我们要开发的是一个 Web 应用，并且导入了专门开发 Web 应用的框架，因此能够轻松实现需求①在 Web 浏览器上显示、⑦经由网络使用以及⑧同时在多台计算机上显示。

从该应用的需求来看，我们需要的功能如下表所示。

功能	说明	需求编号
提交留言功能	显示可输入名字和留言的表单，保存该表单发送的数据	①、②、③
留言显示功能	取出提交留言功能保存的数据（优先提交日期较新的数据）并显示在页面上	④、⑥
Web 应用	在 Web 浏览器上显示，并且可让多台计算机同时经由网络使用该应用	①、⑦、⑧

需求⑤是与页面相关的问题，我们在下一小节学习页面的时候再处理。这里，我们再确认一遍已明确的功能，看看是否能满足需求。

2.4.3　明确必备的页面

接下来我们根据应用的功能分析所需的页面，结果如下。

功能	必备的页面	说明
提交留言功能	可以输入名字和留言的表单	可供输入名字与留言的栏、提交按钮
留言显示功能	显示有名字和留言的列表	在页面上列表显示已提交的名字和留言

需求中提到整个应用由一个页面构成（需求⑤），所以我们要将这两个页面元素糅合到同一个页面中。于是，必备的页面就成了下面这样。

必备的页面	页面元素
提交和显示留言的页面	可以输入名字和留言的表单、显示有名字和留言的列表

至此，我们明确了必备的页面。接下来可以开始设计页面了。

2.5　页面设计

需要的功能和页面都已经敲定，接下来，我们就动手设计页面吧。

至于为什么先实现页面而非功能，是因为当页面完成后，我们可以更好地把握成品应用的整体印象。在把功能植入系统之前，即页面设计阶段，我们需要先写好 HTML 文件和 CSS 文件。

2.5.1　确定成品页面的形式

在开始编写 HTML 代码之前，还需先按捺住自己跃跃欲试的心情，把我们希望呈现的页面明确下来。所以，现在要做的就是把必备页面转化成图。这一步称为页面设计。

建议各位先在纸上画出页面的草图。如果希望后期修改起来方便，或者希望页面草图干净漂亮，还可以考虑用绘图软件或页面设计方面的专业软件来画。

这次我们没有使用工具，而是直接画了一张页面草图（图 2.2）。

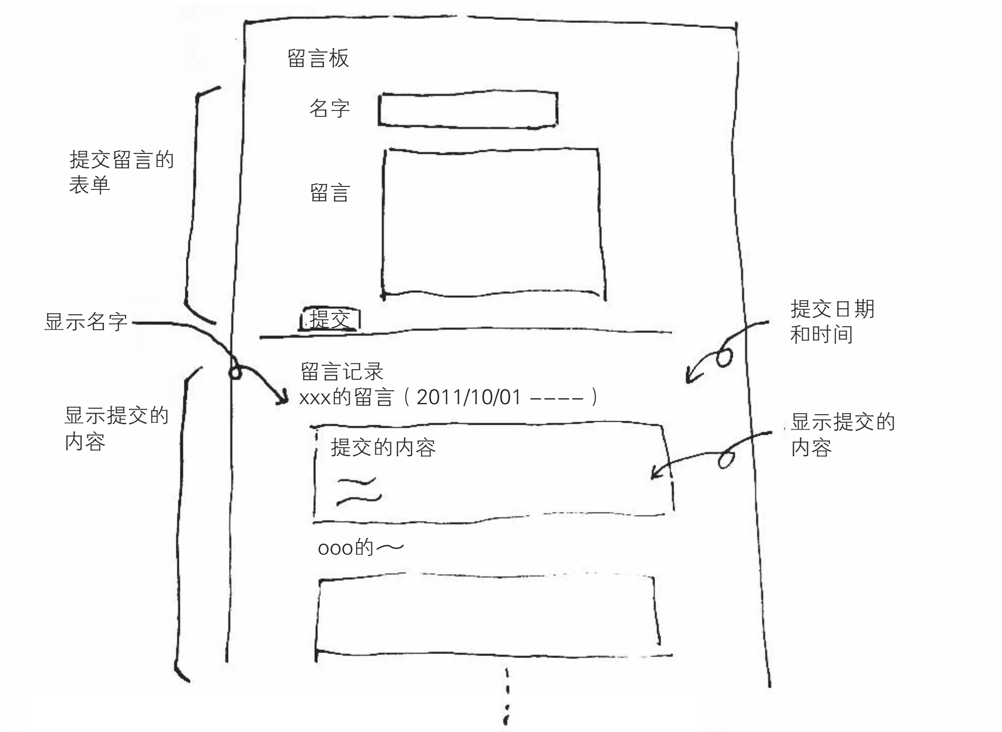


图 2.2 留言板应用的页面草图

按照需求，上面是表单，下面显示提交的评论内容。另外，这张草图还为各个显示元素（表单和文章）添加了注释。这样一张草图不但能在植入功能时为我们提供参考，还能随时提醒我们仍缺少哪些功能，这对开发来说意义重大。

2.5.2 编写 HTML 和 CSS

下面我们开始编写 HTML 文件和 CSS 文件。为了查看其在 Web 浏览器中的效果，我们将在本地环境（主 OS）中进行编写。编写完成的文件可以通过 `scp` 命令发送到服务器，以便后续使用。

首先我们来参考草图编写 HTML 文件。现阶段不必太在意页面布局，这些外观上的东西都可以留在编写 CSS 文件的时候进行调整。

LIST 2.7 是我们编写的 HTML 文件的源码。

LIST 2.7 index.html

```
<!DOCTYPE html>
<html lang="zh">
```

```
  <head>
    <meta charset="utf-8">
    <title>留言板</title>
  </head>
  <body>
    <h1>留言板</h1>
    <form action="/post" method="post">
      <p>请留言</p>
      <table>
        <tr>
          <th>名字</th>
          <td>
            <input type="text" size="20" name="name">
          </td>
        </tr>
        <tr>
          <th>留言</th>
          <td>
            <textarea rows="5" cols="40" name="comment"></textarea>
          </td>
        </tr>
      </table>
      <p><button type="submit">提交</button></p>
    </form>
    <div>
      <h2>留言记录</h2>
      <h3>游客　的留言(2014/10/31 10:00:00):</h3>
      <p>
        留言内容<br>
        留言内容
      </p>
      <h3>游客　的留言(2014/10/31 09:00:00):</h3>
      <p>
        留言内容<br>
        留言内容
      </p>
    </div>
  </body>
</html>
```

用 Web 浏览器打开这个 HTML 文件，结果如图 2.3 所示。

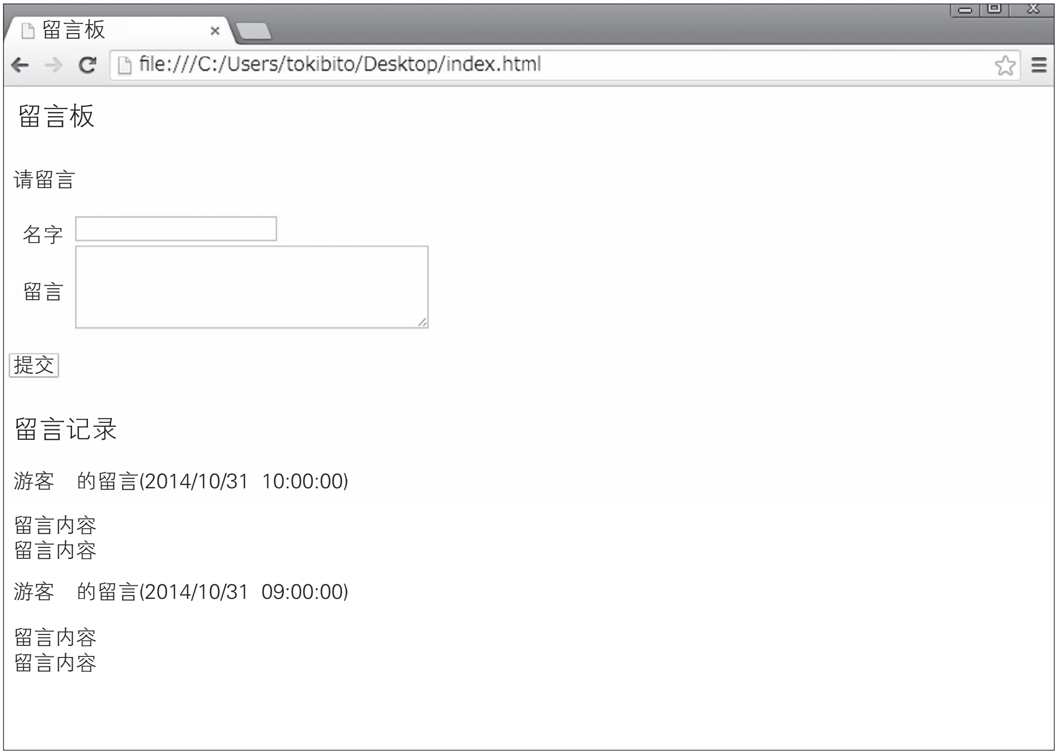


图 2.3　只有 HTML 文件的页面

这样，HTML 文件就写好了，接下来开始写 CSS 文件。此时需要调整字体大小、颜色、显示位置等外观（风格）。

LIST 2.8 是写好的 CSS 文件。

LIST 2.8　main.css

```
body {
  margin: 0;
  padding: 0;
  color: #000E41;
  background-color: #004080;
}
h1 {
  padding: 0 1em;
  color: #FFFFFF;
}
form {
  padding: 0.5em 2em;
  background-color: #78B8F8;
```

```
}
.main {
  padding: 0;
}
.entries-area {
  padding: 0.5em 2em;
  background-color: #FFFFFF;
}
.entries-area p {
  padding: 0.5em 1em;
  background-color: #DBDBFF;
}
```

为了让这个样式表生效，我们需要在 HTML 文件中添加 link 标签读取 CSS，并在相应位置指定 class。修改后的 HTML 文件如 LIST 2.9 所示。

LIST 2.9　index.html

```
<!DOCTYPE html>
<html lang="zh">
  <head>
    <meta charset="utf-8">
    <title>留言板</title>
    <link rel="stylesheet" href="main.css" type="text/css">  <!-- 添加 link 标签 -->
  </head>
  <body>
    <h1>留言板</h1>
    <form action="/post" method="post">
      <p>请留言</p>
      <table>
        <tr>
          <th>名字</th>
          <td>
            <input type="text" size="20" name="name">
          </td>
        </tr>
        <tr>
          <th>留言</th>
          <td>
            <textarea rows="5" cols="40" name="comment"></textarea>
          </td>
        </tr>
      </table>
      <p><button type="submit">提交</button></p>
```

```
    </form>
    <div class="entries-area">  <!-- 在 div 标签中添加 class 属性 -->
      <h2> 留言记录 </h2>
      <h3> 游客　的留言 (2014/10/31 10:00:00):</h3>
      <p>
        留言内容 <br>
        留言内容
      </p>
      <h3> 游客　的留言 (2014/10/31 09:00:00):</h3>
      <p>
        留言内容 <br>
        留言内容
      </p>
    </div>
  </body>
</html>
```

我们用 Web 浏览器打开套用 CSS 后的 HTML 文件，如图 2.4 所示。

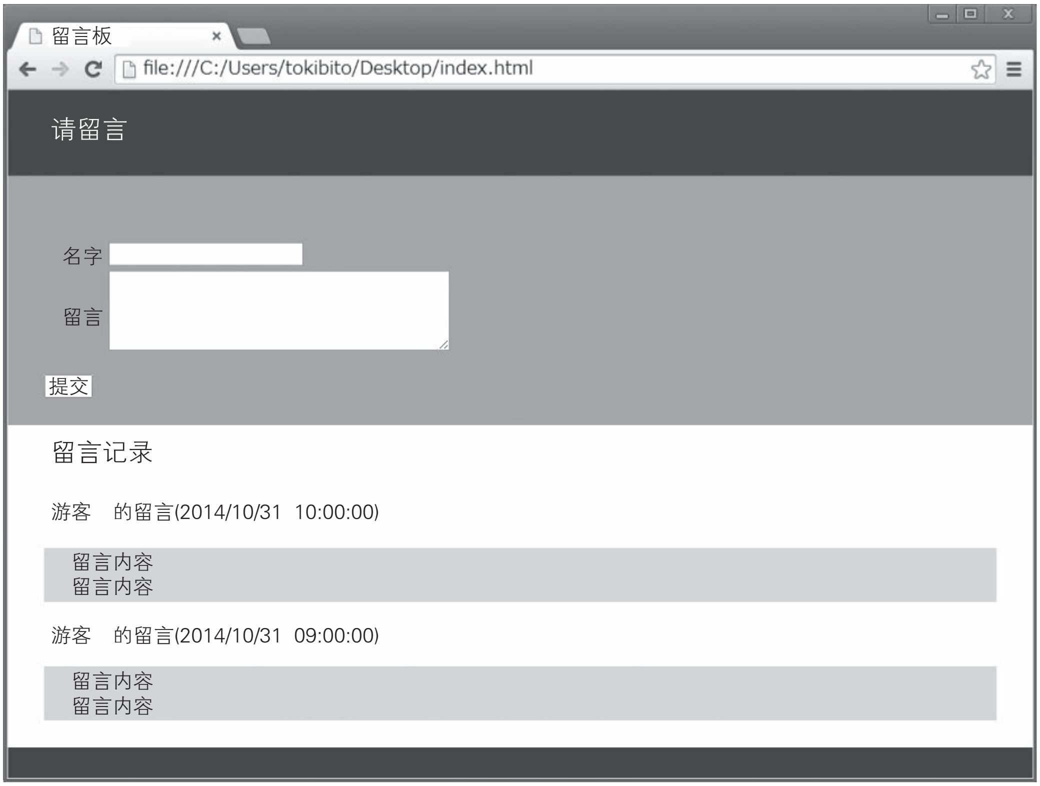


图 2.4　套用 CSS 后的页面

这样我们就写好了显示页面用的 HTML 文件和 CSS 文件。现在用 scp 命令将已写好的文件发送到 VirtualBox 上的 Ubuntu 环境下。发送文件的命令如 LIST 2.10 所示。

LIST 2.10　用 scp 命令发送文件

```
$ scp -P 2222 index.html main.css bpbook@127.0.0.1:
```

NOTE

SSH 连接的设置等问题请参考附录 B。

下一步我们要实现功能，并将其植入页面中。

2.6　实现功能

终于到了编写 Python 程序的阶段。我们准备让服务器端的程序来实现这个应用的必备功能。这里将优先实现比较重要的功能，或者能提高其他部分实现速度的功能。

保存和读取用户提交的数据是这个应用的核心部分，所以我们先从它下手。

2.6.1　保存留言数据

提交功能不但要能保存表单传来的名字和留言，还得能将提交日期及时间存储下来，供显示时使用。

这里我们用 Python 的标准模块 shelve 来存储数据。shelve 能够像 Python 字典对象一样操作数据，将对象持久化。也就是说，shelve 会将提交的数据转换成字典对象，以列表形式保存多个字典，然后将这些字典保存在 shelve 中。

下面来编写留言板的脚本文件。guestbook.py 实现了负责保存数据的 save_data 函数，具体代码如 LIST 2.11 所示。

LIST 2.11　guestbook.py

```
# coding: utf-8
import shelve

DATA_FILE = 'guestbook.dat'

def save_data(name, comment, create_at):
    """ 保存提交的数据
    """
```

```
    # 通过 shelve 模块打开数据库文件
    database = shelve.open(DATA_FILE)
    # 如果数据库中没有 greeting_list，就新建一个表
    if 'greeting_list' not in database:
        greeting_list = []
    else:
        # 从数据库获取数据
        greeting_list = database['greeting_list']
    # 将提交的数据添加到表头
    greeting_list.insert(0, {
        'name': name,
        'comment': comment,
        'create_at': create_at,
    })
    # 更新数据库
    database['greeting_list'] = greeting_list
    # 关闭数据库文件
    database.close()
```

然后再来看看 save_data 函数的运行情况。我们通过终端在 guestbook.py 文件所在的目录下启动 Python shell，然后像 LIST 2.12 这样通过 Python shell 加载并执行 guestbook 模块的 save_data 函数。

LIST 2.12　save_data 函数的运行测试

```
$ ls  # 确认 guestbook.py 位于当前目录下
guestbook.py
$ python  # 启动 Python shell
>>> import datetime
>>> from guestbook import save_data
>>> save_data('test', 'test comment',
...           datetime.datetime(2014, 10, 31, 10, 0, 0))
>>>
```

这里我们将 datetime 模块的日期时间对象传递给传值参数 create_at，以此来保存提交的日期和时间。在这个阶段，我们只能看出它的运行是否报错，至于数据是不是真的被保存下来了，还要等取出数据的功能实现之后才能知道。

2.6.2　获取已保存的留言列表

实际上，我们在保存数据的时候就从 shelve 模块中取出过数据，现在只把这部分代码单独拿出来做成函数即可。

在 guestbook.py 中添加 load_data 函数（LIST 2.13）。

LIST 2.13　guestbook.py

```
def load_data():
    """ 返回已提交的数据
    """
    # 通过 shelve 模块打开数据库文件
    database = shelve.open(DATA_FILE)
    # 返回 greeting_list。如果没有数据则返回空表
    greeting_list = database.get('greeting_list', [])
    database.close()
    return greeting_list
```

这里同样通过 Python shell 查看其运行情况。启动 Python shell，加载函数并运行（LIST 2.14）。

LIST 2.14　load_data 函数的运行测试

```
>>> from guestbook import load_data
>>> load_data()
[{'comment': 'test comment', 'name': 'test', 'create_at': datetime.datetime
(2014, 10, 31, 10, 0)}]
```

如果运行正常，那就表示我们能获取 save_data 函数保存下来的数据。

2.6.3　用模板引擎显示页面

从文件中取出数据之后，为了将其显示到页面上，我们要使用模板引擎。模板引擎可以将模板（程序的雏形）与要植入模板内的数据合并输出。Flask 标准支持 Jinja2 模板引擎。

接下来创建 templates 目录，然后将前面已经写好的 HTML 文件放到该目录下（LIST 2.15）。这样一来，我们就可以以它为模板生成 HTML 了。

LIST 2.15　放置模板

```
$ mkdir templates
$ mv index.html templates/
```

下面从程序端入手，用这个 HTML 文件（模板）来完成页面的显示。先添加代码，让 guestbook.py 调用 Flask，然后再添加用来显示首页的函数以及用来启动 Web 服务器的代码（LIST 2.16）。

LIST 2.16 guestbook.py

```
# coding: utf-8
import shelve

from flask import Flask, request, render_template, redirect, escape, Markup

application = Flask(__name__)

DATA_FILE = 'guestbook.dat'

def save_data(name, comment, create_at):
    """ 保存提交的数据
    """
    # 省略

def load_data():
    """ 返回已提交的数据
    """
    # 省略

@application.route('/')
def index():
    """ 首页
    使用模板显示页面
    """
    return render_template('index.html')

if __name__ == '__main__':
    # 在 IP 地址 127.0.0.1 的 8000 端口运行应用程序
    application.run('127.0.0.1', 8000, debug=True)
```

我们将 Flask 类的实例赋给变量 application，传值参数指定为 __name__ 变量的模块名。方法 route 是一个装饰器，负责注册针对特定 URL 执行的函数。这里我们让主页的 URL 对应执行 index 函数。render_template 函数负责将指定文件用作模板，再通过模板引擎进行输出。

Flask 类的 run 方法用于启动 Web 服务器并执行应用程序，传值参数用来指定要绑定的 IP 地址及端口。另外，将 debug 选项指定为 `True` 时，一旦应用程序出错，Web 浏览器端就会启动可用的调试程序。

接下来用 `python` 命令运行 guestbook.py（LIST 2.17）。

LIST 2.17 运行已开发完毕的 Web 应用

```
$ python guestbook.py
```

```
 * Running on http://127.0.0.1:8000/
 * Restarting with reloader
```

运行 guestbook.py 之后，在该环境的（这里是 VirtualBox 上的 Ubuntu）127.0.0.1 地址的 8000 端口等待请求的应用服务器（Web 服务器）就会启动。我们可以在终端同时按下 Ctrl 键和 C 键，关闭这个服务器。

现在我们需要设置好 8000 端口的端口转发，然后在 Web 浏览器中打开 `http://127.0.0.1:8000/`。如何，看到页面没有？

我们会发现 CSS 文件并没有生效，所以需要重新调整一下 CSS 文件的位置，让它能够被读取。Flask 会公开 static 目录下存放的静态文件。接下来我们创建一个 static 目录，把 main.css 文件放进去（LIST 2.18）。

LIST 2.18　放置静态文件

```
$ mkdir static
$ mv main.css static/
```

另外，还要将 templates/index.html 中的 CSS 文件引用位置改为 /static/main.css（LIST 2.19、LIST 2.20）。

LIST 2.19　templates/index.html（更改前）

```
<link rel="stylesheet" href="main.css" type="text/css">
```

LIST 2.20　templates/index.html（更改后）

```
<link rel="stylesheet" href="/static/main.css" type="text/css">
```

现在再用 Web 浏览器打开该页面，就会发现 CSS 文件这时已经生效了。

接下来，我们把从数据库中取出的内容显示在页面上。修改 guestbook.py 文件，让 index 函数调用 load_data 函数，同时使模板能够使用 load_data 函数取出来的数据（LIST 2.21）。

LIST 2.21　guestbook.py

```
@application.route('/')
def index():
    """ 首页
    使用模板显示页面
    """
    # 读取已提交的数据
    greeting_list = load_data()
    return render_template('index.html', greeting_list=greeting_list)
```

render_template 函数可以将关键字传值参数所指定的值用作模板变量。比如本例就使用了

名为 greeting_list 的模板变量。

此外，我们还要修改模板 templates/index.html，使其能够使用模板变量进行显示。现在对 HTML 中的显示留言部分作如下修改（LIST 2.22）。

LIST 2.22　templates/index.html

```
<div class="entries-area">
  <h2> 留言记录 </h2>
  {% for greeting in greeting_list %}
    <h3>{{ greeting.name }}
        的留言 ({{ greeting.create_at }}):</h3>
    <p>{{ greeting.comment }}</p>
  {% endfor %}
</div>
```

模板内可以使用一种特殊的描述方式，即模板语言。Jinja2 的模板可以通过 {%…%} 的形式使用 if 或 for 等控制语句。植入有模板变量的部分用 {{…}} 的形式描述。在上述模板中，程序会从 greeting_list 中逐一取值并赋给模板变量 greeting，然后使用从 for 到 endfor 之间的模板变量进行循环输出。

这样一来，应用就能将 save_data 函数保存的数据显示在页面上了。

2.6.4　准备评论接收方的 URL

下一步我们用 save_data 函数保存表单提交来的数据。由于模板文件中表单的 action 值为 /post，所以我们就做出这个 URL。

这里，我们将 post 函数添加到 guestbook.py 的 `if __main__ ...` 前面（LIST 2.23）。

LIST 2.23　guestbook.py

```
@application.route('/post', methods=['POST'])
def post():
    """ 用于提交评论的 URL
    """
    # 获取已提交的数据
    name = request.form.get('name')  # 名字
    comment = request.form.get('comment')  # 留言
    create_at = datetime.now()  # 投稿时间（当前时间）
    # 保存数据
    save_data(name, comment, create_at)
    # 保存后重定向到首页
    return redirect('/')
```

在 Flask 中，可以用 request.form 引用表单发来的数据。此外，由于保存数据后需要重新显示首页，所以我们要返回 redirect 函数的结果并进行重定向。

post 函数使用了 datetime 模块，所以需要在文件开头添加如 LIST 2.24 所示代码，以导入该模块。

LIST 2.24　导入 datetime 模块（guestbook.py）

```
# coding: utf-8
import shelve
from datetime import datetime  # 添加此行
```

至此，保存数据的功能也实现了。应用的运行部分基本完工。

2.6.5　调整模板的输出

程序运行所需的功能现在已经基本上都实现了，但这里至少还有两点需要注意。

- 表单提交多行留言时，无法正常显示留言
- 显示的时间精确到了毫秒

要想解决这两个问题需创建一个模板过滤器。模板过滤器会对模板变量的值加以转换并输出。接下来，我们在 guestbook.py 的 `if __main__ ...` 前添加如 LIST 2.25 所示代码，将模板过滤器导入模板。

LIST 2.25　guestbook.py

```
@application.template_filter('nl2br')
def nl2br_filter(s):
    """ 将换行符置换为 br 标签的模板过滤器
    """
    return escape(s).replace('\n', Markup('<br>'))

@application.template_filter('datetime_fmt')
def datetime_fmt_filter(dt):
    """ 使 datetime 对象更容易分辨的模板过滤器
    """
    return dt.strftime('%Y/%m/%d %H:%M:%S')
```

Flask 类的 template_filter 方法是一个装饰器，它负责将函数注册为指定名称的模板过滤器。程序运行时，指定了模板过滤器的模板变量会被作为传值参数传递给相应函数，而函数的返回值则为最后输出的值。在上述源码中，我们注册了 nl2br 和 datetime_fmt 两个模板过滤器。

接下来，我们修改一下 templates/index.html 文件，让模板能够使用这两个模板过滤器（LIST 2.26）。

LIST 2.26 templates/index.html

```
<div class="entries-area">
  <h2> 留言记录 </h2>
  {% for greeting in greeting_list %}
    <h3>{{ greeting.name }}
        的留言 ({{ greeting.create_at|datetime_fmt }}):</h3>
    <p>{{ greeting.comment|nl2br }}</p>
  {% endfor %}
</div>
```

在模板内指定模板过滤器的方法很简单，只需要在模板变量的名称后面加管道符“|”再加模板过滤器名即可。在本次的模板中，我们给 greeting.create_at 指定了 datetime_fmt 过滤器，给 greeting.comment 指定了 nl2br 过滤器。

至此，服务器端的功能、功能与页面的对接已经全部完工。从 Web 浏览器看到的效果如图 2.5 所示。显示留言部分[①]显示了我们已提交的内容。



图 2.5 植入功能后的页面

① 留言记录部分的 tokibito 为本章撰写者冈野真也先生的 Twitter 用户名。——编者注

整个应用的开发过程到这里就结束了，接下来就是确认该应用的运行是否正常，以及看一看成品是否能满足我们当初定下的需求。

2.7　查看运行情况

虽然程序的编写已经结束，但我们还需要确认应用程序能不能按预想的样子运行。下面就来逐条确认刚刚开发完成的应用是否满足需求，以及运行上是否存在问题。到了这一步，可能会有人觉得“开发的时候就已经逐条确认过了，肯定不会有问题的”。这种心情可以理解。但要知道，我们在编码后期阶段所作的修改可能会导致之前的功能出现 Bug，而且也难保开发过程中不会遗漏某些需求。因此要想制作一个品质上乘的应用程序，这项确认工作必不可少。

好了，现在让我们回顾一下这个留言板应用的需求。

① 在 Web 浏览器上显示一个包含“提交留言”表单的页面
② 可以在提交留言表单中输入名字和留言正文
③ 通过提交留言表单发送的名字和留言内容会被保存
④ 已保存的名字、留言、提交日期会显示在页面中
⑤ 整个应用由一个页面构成，页面上部为提交留言表单，下部显示已提交的内容
⑥ 提交的内容按新旧顺序由上到下排列
⑦ 可经由网络（互联网）使用本系统
⑧ 可同时在多台计算机上显示已提交的内容

现在运行开发服务器，查看成品是否能满足这 8 项需求。

只要通过 Web 浏览器访问 `http://127.0.0.1:8000/` 即可查看需求①是否得到了满足。与此同时，由于虚拟机和本地计算机之间经由网络连接，所以⑦和⑧也没有问题。至于②，只要我们能在页面中的表单里填写名字和留言内容，那就是过关了。接下来输入内容并点击提交按钮，随后页面被刷新，刚刚提交的内容显示在了页面上。于是⑤也搞定了。然后重启开发服务器，再次通过 Web 浏览器读取页面，如果之前提交的内容能够正常显示，就表示③和④也 OK。最后我们再进行一次输入和提交，只要后来提交的内容显示在最上方，那么需求⑥也就满足了。

另外，这次我们开发的应用没有对使用者输入的字符串做任何限制，而且输入的内容会直接以植入 HTML 的形式显示出来。因此，接下来需要确认是否存在跨站脚本攻击漏洞。

NOTE

跨站脚本攻击（Cross Site Scripting，XSS[①]）是一种常见的漏洞，用户能在某些以植入HTML 的形式显示输入内容的应用中，故意植入一些攻击型的脚本（比如 HTML 标签或JavaScript 等）。

XSS 漏洞可被用在会话劫持、钓鱼等恶意行为上。

要验证应用中是否含有 XSS 漏洞，只需要像 LIST 2.27 这样，在留言输入栏中键入带有JavaScript 代码的 HTML 标签，然后点击提交即可。

LIST 2.27 验证跨站脚本攻击时需要输入的内容

```
<script>alert('NG')</script>
```

显示提交内容的区域内是否显示出了我们输入的字符串呢？如果该应用含有 XSS 漏洞，那么 NG 字样将显示在浏览器的警告中。

我们在本次开发中使用的是 Jinja2 模板引擎，它在翻译字符串时会自动忽略 HTML 标签的“< 和 >”符号。因此，我们的应用能够成功避免 XSS 漏洞。

建议各位在最后查看运行情况时也检测一下其他可能造成安全问题的漏洞。

NOTE

与 XSS 同样恶名昭彰的漏洞还有跨站请求伪造（Cross Site Request Forgery，CSRF）。用户通过与目标应用程序无关的外部输入表单（这些输入表单通常用于攻击）发送数据，一旦目标应用程序处理了这些数据，就会引发使用者意料之外的操作。

CSRF 漏洞可被用来触发使用者意料之外的操作（比如在线购买商品、泄露个人信息等）。

本章中开发的应用并没有对 CSRF 进行防范。由于 XSS 和 CSRF 这两种攻击手法可以相互组合出新的攻击手法，所以建议各位务必做好防范工作。

如果上述过程全部顺利通过，我们的应用开发就可以宣告完工了。各位辛苦了。

最终的文件结构如下表所示。

文件路径	说明
guestbook.py	服务器程序
guestbook.dat	提交数据文件

① Cross Site Scripting 的缩写为 XSS，这是为了和层叠样式表（Cascading Style Sheet，CSS）有所区分。
——编者注

（续）

文件路径	说明
static/main.css	CSS 文件
templates/index.html	输出 HTML 的模板，用于显示提交 / 留言列表页面

2.8　小结

Web 应用开发的第一步是确定自己要开发什么东西。要是对自己要做的东西没有概念，开发就不可能进行下去。另外，如果必备的页面和功能不够明确，那么编写代码的过程将是相当痛苦的。要想让开发如行云流水般流畅，那就必须将这些不确定的因素统统明确下来，确定一个开发流程。

在本章中，我们强调了 Web 应用是一个发送动态内容的程序，并且用户可以经由 Web 浏览器使用 Web 应用，同时还学习了 Web 浏览器 / 服务器 / 应用之间的基本通信机制。另外，我们还以留言板应用为例，从确定需求到最终完成，边学边练地一起了解了整个应用开发流程。本章学习的开发流程适用于多种应用的开发。

第 3 章　Python 项目的结构与包的创建

在 Python 圈子里，有许多开发者会无偿公开自己开发的程序库。为了让使用者能够通过 pip 命令安装这些库，我们在发布时需要将其创建成一种特殊的文件，这种文件就是程序包。在使用 Python 语言进行开发的过程中，Python 自带的库往往不能满足我们，因此我们还需要用到这些程序包。

本章将介绍程序包的制作流程。首先，我们要了解一下 Python 项目开发环境的相关工具。然后，我们来了解一下该环境下的相对易于掌握的 Python 项目目录结构以及文件结构，同时对第 2 章中开发的留言板应用进行整理，封装成包。最后还将学习如何将我们开发的项目发布在 PyPI 上，与全世界人分享。

3.1　Python 项目

用 Python 开发的应用程序达到一定规模后，必然会出现多个模块（.py）或程序包目录。同时除源码以外，说明性质的文本文件、管理相关程序库的元信息等都会越来越多。这些为同一个目的服务的文件、目录以及元信息，就是我们所说的项目。

实际上，Python 项目的内部结构是因项目而异的。这里，一个完整的结构需要满足以下条件。

- 拥有一个在版本管理之下的源码目录
- 程序信息在 setup.py 中定义
- 在一个 virtualenv 环境中运行

对于 Python 开发上的一些约定俗成的工具来说，满足上述条件的结构更便于处理。我们在第 1 章中介绍的 pip 和 virtualenv 都属于这类工具。

这些 Python 中的约定俗成的工具也随着时代逐渐变化着。近年来 PEP 标准正被逐渐推行。为了规范这些约定俗成的东西，2013 年成立了 PyPA（Python Packaging Authority）工作组。PyPA 负责 Python 封装方面相关工具的维护，以及 PEP 标准化等工作。许多老牌的封装相关工具都被移交给 PyPA 管理，包括本书中使用的 pip、virtualenv、wheel 现在也都由 PyPA 提供。

如今，许多 Python 项目的结构都以 PyPA 提供的工具为参照，选用了适合这些工具的文件、目录结构。

如果项目的结构符合标准，那么它与工具之间就会有很强的亲和力，而且便于今后自己或

其他开发者进一步开发。另外，本章中介绍的结构与流程不但适用于个人的开发环境，同样也适用于团队的开发环境。

> NOTE
>
> 在 PyPA 的封装文档中，将以发布为目的的一个整体单位称为一个项目（Project）。
>
> https://packaging.python.org/en/latest/glossary.html#term-project

3.2　环境与工具

本节，我们将对 Python 项目开发的必备工具进行了解。此外，还将学习项目的目录结构、必备程序包的管理方法以及安装方法。

3.2.1　用 virtualenv 搭建独立环境

如果大量项目全都混杂在一个环境下，程序很可能会在预想不到的地方停止运行，而且不利于我们把握当前环境的具体状态。所以为了防止出现这些恼人的情况，我们建议各位搭建简单的独立环境。

使用 virtualenv 可以给每个项目搭建一个独立的 Python 环境。

独立环境有以下优点。

- 添加程序包以及变更版本时，能将影响控制在当前环境内
- 便于判断已安装的程序包是否可以删除
- 不再需要该环境时，可以直接删除整个环境
- 一旦出了问题，那么问题必然出在该环境的项目上，这就有助于我们找出问题所在

◉ virtualenv

> NOTE
>
> 当前介绍的版本为 virtualenv 1.11.6。

用 pip 安装外部程序库时，该库会被安装到 Python 的安装目录下。比如 Python 是安装在了 /usr/local/ 目录下，那么该外部程序库就会被安装在 /usr/local/lib/python2.7/site-packages 目录下，这就是库的默认安装路径。但这样一来，不同目的的程序库就全都安装到了同一目录下，不但

容易导致版本冲突，而且很难分辨出哪些程序库已经没用了。

另外，在 /usr/local/ 下安装东西时，需要我们提供该目录的写入权限。我们对自己的计算机中的 OS 目录具有写入权限，但是不一定对其他计算机的 OS 目录也拥有写入权限。就算有，乱用权限也很容易导致意外事故，因此应尽量避免乱用权限。

其实，这些问题都可以用 virtualenv 解决。

virtualenv 的主要特征体现在下列功能上。

- 在 virtualenv 环境中可自由安装 Python，不需要提供 OS 管理员权限
- 在 virtualenv 环境下，可根据目的不同来安装程序库，这样一来包的安装目的和依存关系就会更加明确
- 仍然使用 Python 主体，且虚拟环境仅由一小部分备份文件构成，因此环境搭建速度快，占用硬盘空间小
- 无视 Python 主体的 site-packages，而且能分离主体上已安装完毕的程序包
- 可以用 `activate/deactivate` 命令随时启动 / 关闭 virtualenv 环境

virtualenv 命令可以将任意目录设置为“virtualenv 环境（Python 虚拟环境）”。激活 virtualenv 环境之后，Python 解释器会将该目录识别为默认安装目录。所以，如果我们此时用 pip 命令安装程序库，那么这个程序库将被安装到 virtualenv 环境中。

下面我们来了解一下 virtualenv 的一般使用方法以及一些常用选项。其他详细知识请参考以下网站。

> Reference Guide - virtualenv 1.11.6 documentation
> `https://virtualenv.pypa.io/en/latest/`

专栏 Python 标配的用户站点目录功能与 virtualenv 的差异

用户站点目录功能由 PEP 370[①] 提出，随后被 Python 采纳并从 Python 2.6 版本开始提供。这个功能让 Python 解释器能够识别安装在用户目录 $HOME/.local 下的程序库。与此同时，pip 也纳入了这一机制，允许用户使用 --user 选项将程序库安装到 $HOME/.local 目录下。

有了这个功能之后，用户不必使用 virtualenv 就能自由地安装程序库。不过，用户站点目录功能无法像 virtualenv 一样搭建多个环境，自然也就不能在多个环境中切换。

① `https://www.python.org/dev/peps/pep-0370`

◉ virtualenv 的使用方法

接下来我们看一看 virtualenv 的安装、virtualenv 环境的搭建以及启动（LIST 3.1、LIST3.2）。

LIST 3.1　安装 virtualenv

```
$ sudo pip install virtualenv
```

LIST 3.2　virtualenv 环境的搭建及启动

```
$ cd /home/bpbook/work
$ virtualenv venv
$ ls -F
venv/
$ source venv/bin/activate
(venv)$
```

如上所示，使用某一 virtualenv 环境的 `activate` 命令后，该 virtualenv 环境就会被设置为默认的 Python 执行环境。这里我们搭建了名为 venv 的 virtualenv 环境，因此命令提示符前会出现环境名（venv）。

执行 `activate` 后，PATH、PROMPT 等数个环境变量会被改写。这样一来，对象 virtualenv 环境的 bin 目录将在 PATH 搜索中被优先处理。此时，只有环境变量会被修改，文件并不会有任何变动。

在这个状态下，计算机会优先使用 venv/bin 目录下的执行文件来执行各个命令。给 virtualenv 环境安装额外的程序库时，我们需要如 LIST 3.3 所示，在命令提示符前有“(venv)”的状态下执行 pip 命令。

LIST 3.3　给 virtualenv 环境安装程序库

```
(venv)$ pip install requests bottle
(venv)$ pip freeze
bottle==0.12.7
requests==2.4.3
```

如果要使用这个 virtualenv 环境的 Python，则要执行该 virtualenv 环境的 `python` 命令（LIST 3.4）。

LIST 3.4　运行 virtualenv 环境的 Python

```
(venv)$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56) [GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.executable
```

```
'/home/bpbook/work/venv/bin/python'
>>> import requests
>>> import bottle
```

解除 activate 时，需要还原之前被 activate 更改的环境变量。因此我们要结束 shell，或者执行 deactivate 命令（LIST 3.5）。

LIST 3.5 通过 deactivate 命令关闭 virtualenv 环境

```
(venv)$ deactivate
$ python -c "import sys; print sys.executable"
/usr/local/bin/python
```

virtualenv 环境的数量没有上限。这里我们再来搭建一个名为 another-venv 的 virtualenv 环境（LIST 3.6）。

LIST 3.6 搭建另一个 virtualenv 环境

```
$ virtualenv another-venv
$ ls -F
another-venv/   venv/
```

使用 another-venv 环境的 Python 时，我们无法 import 其他 virtualenv 环境安装的库。

LIST 3.7 运行另一个 virtualenv 环境的 Python

```
$ source another-venv/bin/activate
(another-venv)$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56) [GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.executable
'/home/bpbook/work/another-venv/bin/python'
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'requests'
```

可见，每一个 virtualenv 环境都是相互独立运行的 Python 环境（LIST 3.7）。而且每个环境都仅由 Python 的一部分文件构成，所以能够很快搭建完成或者删掉，并不会占用太多硬盘空间。

专栏　virtualenv 环境的硬盘使用量

在 Ubuntu 14.04 下，Python 主体的硬盘使用量大约为 100 MB，每个 virtualenv 环境的硬盘使用量约为 6 MB。不过，如果使用了我们后面即将了解的 --always-copy 选项，那么每个 virtualenv 环境将占用约 50 MB。

不再使用的 virtualenv 环境可以连同其所在目录一起删除（LIST 3.8）。

LIST 3.8　删除 virtualenv 环境

```
(another-venv)$ deactivate
$ rm -R another-venv
```

专栏　在不 activate 的状态下执行 virtualenv 环境的命令

我们执行 `activate` 命令启动 virtualenv 环境时，环境变量会被更新，此后执行命令时会优先使用 venv/bin/ 目录下的文件。实际上，即便是在没有执行 `activate` 命令的状态下，只要我们用完整路径执行了 venv/bin/ 下的命令，就可以运行 virtualenv 环境中的程序。

```
$ python -c "import sys; print sys.executable"
/usr/local/bin/python

$ venv/bin/python -c "import sys; print sys.executable"
/home/bpbook/work/venv/bin/python

$ venv/bin/python

>>> import requests            # 可以从venv环境中导入

$ venv/bin/pip install flask   # 安装到venv环境中
...
```

因此，当我们想有计划地执行 venv 环境的程序，或者想启动服务器时，只要用完整路径指定 venv 环境的程序，就可以在不执行 `activate` 命令的状态下完成运行了。

◉ virtualenv 的选项

virtualenv 为用户提供了许多可用选项。我们可以通过 `virtualenv --help` 来查看选项列表。现在我们选其中一些比较好用的选项来了解一下。

- -p、--python

  指定virtualenv环境下使用的Python，格式如`--python=/usr/local/bin/python2.7`。

如果省略该选项，则默认选择执行virtualenv命令的Python。

- --system-site-packages

  使用当前Python主体上已经安装的程序库。如果省略该选项，则默认忽略Python主体上已经安装的程序库。

- --always-copy

  无论符号链接是否可用，一概不使用符号链接，而是直接复制文件。即便是允许使用符号链接的OS，在某些文件系统下仍会出现符号链接不可用的情况。此时就可以用到这个选项。

- --clear

  删除指定的virtualenv环境中安装的依赖库等，初始化环境。

- -q、-quiet

  执行virtualenv命令时，控制向工作台输出的信息量。

- -v、-verbose

  执行virtualenv命令时，增加向工作台输出的信息量。

LIST 3.9　virtualenv 的选项示例

```
$ virtualenv -q --system-site-packages -p /usr/local/bin/python2.7 venv
```

如果我们经常用到某些选项，可以事先将其写入设置文件。这样一来，我们就不用每次执行命令时都再写一遍了。

在 Linux 系统下，设置文件默认使用 $HOME/.virtualenv/virtualenv.ini。举个例子，如果要在 virtualenv.ini 中指定 LIST 3.9 里的选项，我们可以按照 LIST 3.10 进行描述。

LIST 3.10　virtualenv.ini

```
[virtualenv]
python = /usr/local/bin/python2.7
quiet = true
system-site-packages = true
```

另外，还可以在环境变量中指定选项。如果我们同时用环境变量和设置文件指定了某个选项，那么将以环境变量的指定为准。当然，命令行传值参数的指定优先于一切。

环境变量名是根据选项名自动生成的。将选项名的“--”后的字母全改为大写，短线改为下划线，然后在开头加上 VIRTUALENV_，就是该选项所对应的环境变量。请各位在设置环境变量时记住这个规律。

- **示例**

```
--quiet                  -> VIRTUALENV_QUIET=true
--python                 -> VIRTUALENV_PYTHON=/usr/local/bin/python2.7
--system-site-packages   -> VIRTUALENV_SYSTEM_SITE_PACKAGES=true
```

专栏　--system-site-packages 选项

virtualenv 环境不会引用 Python 主体上安装的程序包，也就是 /usr/local/lib/python2.7/site-packages 目录下的程序包。因此，当我们新建了一个 virtualenv 环境时，这个环境处于干净的初始状态，没有安装任何多余的包。

相反地，如果我们想在 virtualenv 环境中使用 Python 主体上安装的程序包，那么在新建环境时可以加上 --system-site-packages 选项。

不过，一旦加上这个选项，我们就相当于同时使用两个环境，因此很难分辨出程序包在哪里，以及正在使用的是哪个程序包。这会导致 virtualenv 环境的优势大打折扣。所以，除非迫不得已，建议各位尽量不要使用 --system-site-packages 选项。

3.2.2　用 pip 安装程序包

NOTE

我们使用的是 pip 1.5.6。

给 virtualenv 环境额外安装程序包时，需要用该环境的 pip 命令。

pip 是用来安装程序包的命令。既可以经由网络进行安装，也可以直接从本地的程序包文件进行安装。第三方的程序库发布在 PyPI 上。pip 命令会默认从 PyPI 上搜索并安装程序包。

pip 提供了多个子命令。下面便是其子命令列表。

install	安装程序包（指定包名、包文件名、URL 等）
uninstall	卸载程序包
freeze	以 requirements 格式列表输出当前已安装的程序包及其版本
list	列表显示当前已安装的程序包
show	列表显示当前已安装程序包的版本信息
search	按指定关键字在 PyPI 上搜索程序包并显示结果列表
wheel	根据指定 requirement 构建 wheel 文件
help	显示帮助

接下来，我们将对 pip 的选项以及安装、卸载方法进行学习。其他详细内容请参考以下网站。

> Reference Guide - pip 1.5.6 documentation
> https://pip.pypa.io/en/latest/reference/index.html
> User Guide - pip 1.5.6 documentation
> https://pip.pypa.io/en/latest/user_guide.html#configuration

◉ pip 的选项

pip 的选项有两种，一种是不依赖于子命令的共通选项，另一种是指定给子命令的选项。比如 --quiet 和 --proxy 就是所有命令共通的选项，而 install 子命令则拥有 --upgrade 等自己独有的选项（LIST 3.11）。

这两种选项都需要在执行时通过命令行指定。

LIST 3.11 pip 的选项示例

```
$ pip --quiet --proxy=server:9999 install --upgrade requests
```

常用的选项可以事先写在设置文件中。这样可以省去每次都写的麻烦。

Linux 的默认设置文件为 $HOME/.pip/pip.conf。比如我们要在 pip.conf 中指定 LIST 3.11 pip 的选项示例中的选项，则可以按照 LIST 3.12 进行描述。

LIST 3.12 pip.conf

```
[global]
quiet = true
proxy = server:9999

[install]
upgrade = true
```

> **NOTE**
>
> 上述内容只是个例子，并非推荐设置。各位请严格按照自己的需要设置各个选项。

另外，还可以在环境变量中指定选项。如果我们同时用环境变量和设置文件指定了某个选项，那么将以环境变量的指定为准。当然，命令行传值参数的指定优先于一切。

环境变量名是根据选项名自动生成的。将选项名的“--”后的字母全改为大写，短线改为下划线，然后在开头加上 `PIP_`，就是该选项所对应的环境变量。请各位在设置环境变量时记住这个规律。

- 示例

```
--quiet    -> PIP_QUIET=true
--proxy    -> PIP_PROXY=server:9999
--upgrade  -> PIP_UPGRADE=true
```

专栏　关于指定 HTTP 代理

pip 需要使用 HTTP 协议从外部网站获取程序包。在某些企业或环境下，我们需要经由代理才能访问到外部网站。而且有些时候，我们必须输入 ID 和密码才可以使用代理访问外部网站（认证代理）。

在这类环境下使用 pip 时，我们要用 --proxy 选项指定代理，格式为 `[user:passwd@]proxy.server:port`（LIST 3.13、LIST 3.14）。

LIST 3.13　指定 pip 的代理

```
$ pip --proxy=proxy.example.com:1234 install requests
```

LIST 3.14　指定 pip 的认证代理

```
$ pip --proxy=beproud:passwd@proxy.example.com:1234 install requests
```

我们也可以像 LIST 3.15 这样，在设置文件或环境变量中指定 --proxy 选项。

LIST 3.15　在 PIP_PROXY 环境变量中指定

```
$ export PIP_PROXY=proxy.example.com:1234
$ pip install requests
```

除 PIP_PROXY 环境变量外，我们还可以在 HTTP_PROXY 环境变量中指定 Proxy（LIST 3.16）。

LIST 3.16　在 HTTP_PROXY 环境变量中指定

```
$ export HTTP_PROXY=proxy.example.com:1234
$ pip install requests
```

◉ 安装程序包

`pip install` 是安装程序包的命令。install 子命令可以详细指定“安装什么”“从哪里安装”“如何安装”。因此，我们在看帮助文档时会发现它有很多选项和对象指定方法（LIST 3.17）。

LIST 3.17　pip install 的帮助

```
$ pip install -h

Usage:
```

```
  pip install [options] <requirement specifier> ...
  pip install [options] -r <requirements file> ...
  pip install [options] [-e] <vcs project url> ...
  pip install [options] [-e] <local project path> ...
  pip install [options] <archive url/path> ...

...
```

从 PyPI 安装程序包的方法如 LIST 3.18 所示。这里，我们可以指定一个或多个程序包。

LIST 3.18　从 PyPI 安装

```
$ pip install requests
$ pip install flask bottle
```

如果手边有源码，还可以像 LIST 3.19 这样进行安装。

LIST 3.19　从源码包安装

```
$ pip install ./logfilter-0.9.2
```

像上例这样从本地目录安装时，我们需要写明本地路径，如 `./logfilter-0.9.2` 或 `file://logfilter-0.9.2` 等。如果只写 `pip install logfilter-0.9.2`，计算机会跑到 PyPI 上去找名为 `logfilter-0.9.2` 的程序包。

从源码安装时，我们可以使用 -e（--editable）选项。这样一来，安装时不会复制源码，而是直接在该目录下原地进行安装。对于一些尚在开发中的源码，用 editable 进行安装可以省去每次修改代码后重新安装的麻烦（LIST 3.20）。由于程序可以在运行时直接使用我们编辑过的代码，所以 editable 也被称为可编辑安装。

LIST 3.20　指定 editable 安装源码目录

```
$ pip install -e ./logfilter-0.9.2
```

将版本库 clone 下来并进行安装时，需要在版本库的 URL 前添加例如 `hg+` 等版本库类别，然后再执行 `pip install`（LIST 3.21）。在执行本例的过程中，需要在内部使用 `hg` 命令，因此必须有一个可以使用 `hg` 命令的环境。对象为 git 版本库时请将上述 `hg` 替换为 `git`。

现在让我们来执行安装。

LIST 3.21　从 hg 上 clone 并安装

```
$ pip install hg+https://bitbucket.org/shimizukawa/logfilter
```

如果想在 clone 来的源码上进行开发，我们可以同时加上 -e 选项。此时需要如 LIST 3.22 所示，在 URL 末尾添加 `#egg=<<` 程序包名 `>>`。

LIST 3.22　从 hg 上 clone 并 editable 安装

```
$ pip install -e hg+https://bitbucket.org/shimizukawa/logfilter#egg=logfilter
```

如果要一次安装多个程序包，比较简便的方法就是使用对象程序包以及描述这些程序包的 requirements 格式的文件。文件名一般使用 requirements.txt。我们可以自己动手准备这个文件，不过一般我们都会用 `pip freeze` 来输出。requirements.txt 文件用 -r（--requirement）选项指定（LIST 3.23）。

LIST 3.23　通过 requirements.txt 安装

```
$ pip install -r requirements.txt
```

使用 pip 时，如果之前已经安装过指定的程序包，计算机并不会自动将其更新为新版本。指定更新到某一版本需要用 -U（--upgrade）选项（LIST 3.24）。

LIST 3.24　用 -U 选项更新版本

```
$ pip install -U requests
```

如果不想每次都从 PyPI 下载程序包，我们可以用 --download-cache 选项指定缓存目录（LIST 3.25）。此时，为了检查版本，计算机仍会进行网络通信。如果缓存目录中有我们要用的版本，计算机则会直接用缓存文件进行安装。

LIST 3.25　指定下载缓存

```
$ pip install --download-cache=~/.pip-cache requests
```

如果想一直使用缓存目录，建议各位在环境变量中设置该选项（LIST 3.26）。只要维持环境变量中的这一设置，我们就能有效运用缓存。

LIST 3.26　在环境变量中设置下载缓存

```
$ export PIP_DOWNLOAD_CACHE=~/.pip-cache
$ pip install requests
```

NOTE

从 pip-6.0 起，下载缓存改为默认有效。指定 --download-cache 选项时会出现无效警告。

◉ 记录程序包一览表

`pip freeze` 命令用来将已安装程序包及其版本的一览表以 requirements 格式输出。其执行如 LIST 3.27 所示。

LIST 3.27 pip freeze

```
$ pip freeze
Flask==0.10.1
Jinja2==2.7.3
MarkupSafe==0.23
Werkzeug==0.9.6
argparse==1.2.1
itsdangerous==0.24
wsgiref==0.1.2
```

NOTE

从 pip-6.0 起，不再显示 Python 标准库 argparse 和 wsgiref。

`pip freeze` 以 requirements 格式输出程序包及其版本的列表。为了方便 `pip install` 使用这些内容，它们会被保存在 requirements.txt 文件中（LIST 3.28）。这个文件名并不是硬性要求，但大多数情况下我们习惯使用 requirements.txt。

LIST 3.28 将 pip freeze 的输出保存在 requirements.txt 中

```
$ pip freeze > requirements.txt
```

这样一来，我们就可以用类似 `pip install -r requirements.txt` 的方式在其他环境中安装相同版本的程序包了。

如果环境中安装的程序包发生变动，我们可以再次执行 `pip freeze > requirements.txt`，以更新文件内容。此时，用 -r（--requirement）选项指定原来的 requirements.txt 可以确认更新前后的差别（LIST 3.29）。用这种方法能帮助我们确认是否存在意外更改。

LIST 3.29 pip freeze -r 的结果

```
$ pip install bottle
$ pip freeze -r requirements.txt
Flask==0.10.1
Jinja2==2.7.3
MarkupSafe==0.23
Werkzeug==0.9.6
argparse==1.2.1
itsdangerous==0.24
wsgiref==0.1.2
## The following requirements were added by pip --freeze:
bottle==0.12.7
```

专栏　在 requirements.txt 内指定版本库

如果我们在当前环境下用 -e（--editable）安装了版本库的代码，那么执行 `pip freeze` 后，输出结果如下。

```
$ pip install -e hg+https://bitbucket.org/shimizukawa/logfilter#egg=logfilter
$ pip freeze
-e hg+https://bitbucket.org/shimizukawa/logfilter@96fd26dae42053c015d3285c
002d45aa5fe6e324#egg=logfilter-dev
Flask==0.10.1
Jinja2==2.7.3
MarkupSafe==0.23
Werkzeug==0.9.6
itsdangerous==0.24
wsgiref==0.1.2
```

该版本库特定版本的代码会被安装到使用 requirements.txt 进行 `pip install` 时的环境中。所以各位请注意，安装时会执行版本库 clone 操作，而且无论版本库是否接到了新的提交，安装时都会使用我们指定的版本。

◉ 卸载程序包

`pip uninstall` 命令用于卸载、删除当前已安装的程序包。其执行方式如 LIST 3.30 所示。

LIST 3.30　pip uninstall

```
$ pip uninstall flask
Uninstalling flask:
  /home/bpbook/work/venv/lib/python2.7/site-packages/...
  ...
Proceed (y/n)? y
  Successfully uninstalled flask
```

屏幕上会显示待删除文件的列表，并向用户询问是否真的要删除。此时输入 y 并按下 Enter 键即可完成删除操作。

如果不需要确认，可以加上 -y（--yes）选项，其执行如 LIST 3.31 所示。

LIST 3.31　pip uninstall -y

```
$ pip uninstall -y flask
```

请注意，`pip uninstall` 命令只会卸载我们指定的程序包。比如安装 Flask 时会捆绑安装 Jinja2 等 4 个关联程序包，这些程序包在 Flask 被卸载之后仍会残留在环境中。如果我们不再需

要这些程序包，就必须通过 `pip uninstall` 手动删除它们，或者直接重建 virtualenv 环境，选我们需要的程序包重新安装。

3.2.3 小结

开发 Python 项目时，我们可以用 virtualenv 搭建该项目专用的虚拟环境。有了这个环境，我们使用程序库时就不会影响到其他项目了。项目所需的程序库用 pip 进行安装。用 `pip freeze` 命令搭建某项目环境的 requirements.txt 文件之后，我们能够轻松重建该项目所需的环境。除此之外，部署等方面也有 pip 和 virtualenv 的用武之地。这两个工具是 Python 开发环境的基础，建议各位牢记它们的使用方法。

3.3 文件结构与发布程序包

编写完程序之后，我们还要面对从封装成包到发布的复杂流程。在本部分中，我们会尝试把第 2 章中开发的留言板应用放到 PyPI 上进行公开，并在此过程中学习一下 setup.py 的写法以及如何向 PyPI 上传程序包等。

3.3.1 编写 setup.py

首先我们来了解一下 setup.py 的功能。Python 的封装离不开 setup.py。将开发完毕的程序封装成包，可以方便其他用户或其他项目拿去用。而封装的绝大部分时间都要消耗在编写 setup.py 上。

我们用 setup.py 来设置 Python 程序包的信息（元数据）、定义程序包。setup.py 这个文件名是 Python 中定好了的，不可以更改。我们要在这个文件中定义程序包名称、包及依赖包的信息等元数据。

有了 setup.py 之后，我们便能在命令行中执行诸如 `python setup.py sdist` 和 `python setup.py install` 等与包相关的操作了。另外，将程序包注册到 PyPI 的操作也需要通过 setup.py 进行。

◉ setup.py 的命令

首先，我们来做出一个能运行的 setup.py 空壳（LIST 3.32）。

LIST 3.32 setup.py

```
from setuptools import setup
setup(name='guestbook')
```

现在计算机已经可以执行 setup.py 的命令了。各位可以通过 --help-commands 选项查看 setup.py 提供的命令（LIST 3.33）。我们在下面列举了一些有代表性的命令。命令后面的英文注释已经译为了中文。

LIST 3.33　setup.py 的指令一览

```
$ python setup.py --help-commands
标准命令
  build             构建安装所需的全部内容
  clean             删除 'build' 命令创建的所有临时文件
  install           安装 build 目录下的全部内容
  sdist             创建源码包（以 tar、zip 等格式）
  register          将程序包注册到 Python Package Index
  bdist             创建二进制包
  bdist_dumb        创建 'dumb' 格式的二进制包
  bdist_rpm         创建 RPM 格式的二进制包
  bdist_wininst     创建面向 MS Windows 的安装包
  upload            将二进制包上传至 PyPI
  check             检查程序包的设置值是否正确

扩展命令
  develop           以开发模式安装程序包
  setopt            在 setup.cfg 等文件中记录一个选项
  saveopts          将给定的多个选项记录在 setup.cfg 等文件中
  upload_docs       向 PyPI 上传文档
  alias             定义快捷命令
  bdist_egg         创建 'egg' 格式的程序包
  test              原地构建后运行 Unit Test
```

下面我们来创建最基本的源码包。源码包需要通过 `python setyp.py sdist` 命令创建（LIST 3.34）。

LIST 3.34　python setup.py

```
$ python setup.py sdist
running sdist
running egg_info
-（中间省略）-
warning: sdist: standard file not found: should have one of README, README.rst,
 README.txt

running check
warning: check: missing required meta-data: url
```

```
warning: check: missing meta-data: either (author and author_email) or (maintai
ner and maintainer_email) must be supplied

creating guestbook-0.0.0
-（中间省略）-
creating dist
Creating tar archive
removing 'guestbook-0.0.0' (and everything under it)

$ ls dist/
guestbook-0.0.0.tar.gz
```

现在，dist 目录下已经生成了 guestbook-0.0.0.tar.gz 文件。这个 tar.gz 文件目前只包含 setup.py。

另外，我们在执行过程中看到了几个 warning。这些 warning 指出的项目最好都设置一下。后面我们会学习如何进行设置。

3.3.2 留言板的项目结构

首先，我们来了解一下 Python 项目一般的目录结构。当封装对象只有一个“.py”文件时，其结构如 LIST 3.35 所示。

LIST 3.35 项目内只有一个文件时的结构示例

```
/home/bpbook/projectname
   +-- MANIFEST.in
   +-- README.rst
   +-- packagename.py
   +-- setup.py
```

如果封装对象目录下包含多个“.py”或模板等文件，则结构如 LIST 3.36 所示。

LIST 3.36 项目内含多个文件时的结构示例

```
/home/bpbook/projectname
   +-- MANIFEST.in
   +-- README.rst
   +-- packagename/
   |    +-- __init__.py
   |    +-- module.py
   |    +-- templates/
   |         +-- index.html
   +-- setup.py
```

关于这方面，我们的留言板应用由下述文件组成。

文件路径	说明
guestbook.py	服务器程序
guestbook.dat	提交数据文件
static/main.css	CSS 文件
templates/index.html	输出 HTML 的模板，用于显示“提交 / 留言列表”的页面

虽然“.py”文件只有一个，但 static 和 templates 目录下都包含文件。由于我们介绍的前一种项目结构无法安装模板等文件，因此这里需要使用后一种项目结构。

文件最终的安排如 LIST 3.37 所示。

LIST 3.37　留言板项目的目录结构

```
/home/bpbook/guestbook/
   +-- LICENSE.txt
   +-- MANIFEST.in
   +-- README.rst
   +-- guestbook
   |    +-- __init__.py
   |    +-- static/main.css
   |    +-- templates/index.html
   +-- setup.py
```

现在我们来创建 guestbook 目录，将 guestbook.py 文件移动到该目录下并重命名为“__init__.py”（init 前后各两个半角下划线）。另外，templates 和 static 目录也要移动到 guestbook 目录下。guestbook.dat 不是我们要发布的东西，所以这里不需要它。

接下来，我们来实际应用这个封装用的结构。

3.3.3　setup.py 与 MANIFEST.in——设置程序包信息与捆绑的文件

接下来我们将在 setup.py 中设置程序包的信息，然后在 MANIFEST.in 中指定捆绑的文件。那么，我们先来按照顺序了解一下。

◉ setup.py

首先，我们像 LIST 3.38 这样描述 guestbook 项目的 setup.py。

LIST 3.38　最低限度内容的 setup.py

```
from setuptools import setup, find_packages

setup(
    name='guestbook',
```

```
    version='1.0.0',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[
        'Flask',
    ],
)
```

如果一个环境能使用 pip，那么该环境中一定安装了 setuptools 库。虽然用 from distutils.core import setup 这种 Python 标准写法也没有问题，但一般情况下我们习惯使用 setuptools 提供的含有扩展功能的 setup 函数。

下面来了解一下各个参数的意义。

- name

  程序包的名称。这里我们定为'guestbook'。一般情况下，包名都与项目名称一致。但是，用于发布的程序包需要有一个独特的名称，以防止与其他程序包名撞车。实际上，guestbook这个名称实在不够独特。因此，如果一定要使用这个名称，最好在前面加上组织名等，例如beproud.guestbook。

- version

  代表版本号的字符串。这里我们定为'1.0.0'。

- packages

  指定所有捆绑的Python程序包（可以用python命令import的目录名）。举个例子，如果一个项目包含多级目录，那么我们需要用下例所示的方法，列表指定所有程序包。

  ```
  packages=[
      'guestbook', 'guestbook.server', 'guestbook.server.dir',
      'guestbook.storage', ...
  ],
  ```

  find_packages()函数可以自动搜索当前目录下的所有Python程序包并返回程序包名。有了它，我们便可以省去一个个列举的麻烦。

NOTE

如果项目仅由一个“.py”文件构成，那么要用 py_modules 代替 packages 传值参数，在 py_modules 中指定对象模块名。

- include_package_data

  在packages指定的Python包（目录）中，除“.py”之外的文件都称为程序包资源。这个

设置用来指定是否安装Python包中所含的程序包资源。

这里我们要安装templates和static这两个程序包资源，所以将它们指定为`True`。

不过，这一设置并不能将程序包资源与我们要发布的程序包捆绑在一起。捆绑的方法将在MANIFEST.in中学习。

- install_requires

  列表指定依赖包。留言板应用要依赖Flask，所以我们在这里指定Flask。与requirements.txt不同，这里一般不指定版本。

◉ MANIFEST.in

为将 HTML 文件、CSS 文件等程序包资源与程序包捆绑在一起，我们需要用 MANIFEST.in 来指定封装对象文件。

这里我们在 setup.py 所在的目录下创建 MANIFEST.in 文件，指定封装对象文件的范围（LIST 3.39）。

LIST 3.39　MANIFEST.in

```
recursive-include guestbook *.html *.css
```

recursive-include 表示捆绑指定目录下所有与指定类型一致的文件。以 LIST 3.39 为例，我们捆绑了 guestbook 目录下所有与“*.html”和“*.css”一致的文件。

现在我们希望使用这个程序包的环境能安装这些捆绑好的程序包资源。为此，我们需要将前面提到的 install_package_data 指定为 `True`，这一点千万不能忘。

MANIFEST.in 还可以指定捆绑 guestbook 应用不使用的非程序包资源文件，比如 LICENSE.txt。在发布程序包时最好把许可文件也捆绑进去。

假设我们使用了 BSD 许可，并在 LICENSE.txt 文件中描述了许可条款。接下来，我们需要在 MANIFEST.in 里添加对它的捆绑指定（LIST 3.40）。

LIST 3.40　MANIFEST.in

```
recursive-include guestbook *.html *.css
include LICENSE.txt
```

`include` 会捆绑所有与指定类型一致的文件。所以添加了 LIST 3.40 中所示的指定语句后，LICENSE.txt 文件就和程序包捆绑在了一起。另外，我们要安装的是 guestbook 目录，而 LICENSE.txt 文件并不在该目录下，所以 LICENSE.txt 文件并不会被安装到使用该程序包的环境中。

MANIFEST.in 有许多种描述方式，不但可以将某个扩展名的文件全部捆绑起来，还可以剔

除特定扩展名的全部文件。MANIFEST.in 的详细描述方法请查阅 Python 的参考手册。

> Creating a Source Distribution - Python 2.7.12 documentation
> `https://docs.python.org/2.7/distutils/sourcedist.html`

◉ 确认运行情况

为查看前面的设置是否正确，我们需要搭建一个用来开发程序包的 virtualenv 环境并安装该程序包。这里我们用“.venv”作为 virtualenv 环境的目录名（LIST 3.41）。

LIST 3.41　搭建 virtualenv 环境及安装

```
$ cd ..
$ virtualenv .venv
```

此时的目录结构如 LIST 3.42 所示。

LIST 3.42　目录结构

```
/home/bpbook/guestbook/
   +-- .venv/
   +-- LICENSE.txt
   +-- MANIFEST.in
   +-- guestbook
   |    +-- __init__.py
   |    +-- static/main.css
   |    +-- templates/index.html
   +-- setup.py
```

然后我们启动 virtualenv 环境并执行安装。在安装时请加上 -e（--editable）选项进行原地安装（在原目录下直接转为安装状态）（LIST 3.43）。这样一来，我们在开发过程中就不用每改一次都重新安装一遍了。

LIST 3.43　搭建 virtualenv 环境及 editable 安装

```
$ source .venv/bin/activate
(.venv)$ pip install -e .
Obtaining file:///home/bpbook/guestbook
  Running setup.py (path:/home/bpbook/guestbook/setup.py) egg_info for package
from file:///home/bpbook/guestbook

Installing collected packages: guestbook
-（中间省略：安装依赖包）-
  Running setup.py develop for guestbook
```

```
    Creating /home/bpbook/.venv/lib/python2.7/site-packages/guestbook.egg-link
 (link to .)
    Adding guestbook 1.0.0 to easy-install.pth file

    Installed /home/bpbook/guestbook

-（中间省略）-
Successfully installed guestbook
Cleaning up...

(.venv)$ pip freeze
Flask==0.10.1
Jinja2==2.7.3
MarkupSafe==0.23
Werkzeug==0.9.6
guestbook==1.0.0
itsdangerous==0.24
```

现在，guestbook-1.0.0 已经安装到 virtualenv 环境中了。我们可以看到，记录程序包元数据位置的 guestbook.egg-link 文件被安装到 virtualenv 环境中了。easy-install.pth 文件中添加了 guestbook 的源码位置。另外，Flask 及其相关程序包也都安装好了。

这样一来，我们在其他 PC 或服务器上构建环境时，就不必再去一个个地安装依赖包了。如果今后需要添加或更改依赖库，各位只要按照前面讲的流程更新 setup.py，然后再执行一次 `pip install` 即可。

3.3.4　setup.py——创建执行命令

我们在第 2 章开发的留言板是一个直接从 Python 启动的脚本。要想让下载它的人用起来更方便，那最好生成一些用户命令。这里我们通过设置 setup.py，让其自动生成 `guestbook` 命令（LIST 3.44）。

LIST 3.44　让 setup.py 生成命令

```
from setuptools import setup

setup(
    name='guestbook',
    version='1.0.0',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[
```

```
        'Flask',
    ],
    entry_points="""
        [console_scripts]
        guestbook = guestbook:main
    """,
)
```

我们在 setup.py 中添加了 entry_points。这样一来，在安装程序包时就会自动生成 `guestbook` 命令。用户执行 `guestbook` 命令时将会调用 guestbook 模块的 main 函数。

但是 guestbook/__init__.py 中还没有 main 函数，所以我们需要添加这个函数，具体代码如 LIST 3.45 所示。

LIST 3.45 guestbook/__init__.py

```
    ⋮
    ⋮

def main():
    application.run('127.0.0.1', 8000)

if __name__ == '__main__':
    # 在 IP 地址 127.0.0.1 的 8000 端口运行应用程序
    application.run('127.0.0.1', 8000, debug=True)
```

然后我们再次执行安装命令，看看是否能生成 `guestbook` 命令。

即便是在 editable 安装的状态下，如果想反映出对元信息进行的修改（比如添加命令、更改依赖库等），也需要重新执行一次安装命令（LIST 3.46）。

LIST 3.46 重新安装

```
(.venv)$ pip install -e ./guestbook
-（中间省略）-
Successfully installed guestbook
Cleaning up...

(.venv)$ ls .venv/bin/guestbook
guestbook

(.venv)$ guestbook
 * Running on http://127.0.0.1:8000/
 * Restarting with reloader
```

可以看到，`guestbook` 命令已经成功生成，而且可以正常运行了。

3.3.5　python setup.py sdist——创建源码发布程序包

创建用于发布的程序包时，需要如 LIST 3.47 所示，执行 `python setup.py sdist` 命令。

LIST 3.47　python setup.py sdist

```
$ python setup.py sdist
running sdist
running egg_info
writing requirements to guestbook.egg-info/requires.txt
writing guestbook.egg-info/PKG-INFO
writing top-level names to guestbook.egg-info/top_level.txt
writing dependency_links to guestbook.egg-info/dependency_links.txt
reading manifest file 'guestbook.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'guestbook.egg-info/SOURCES.txt'
running check
creating guestbook-1.0.0
-（中间省略）-
making hard links in guestbook-1.0.0...
hard linking LICENSE.txt -> guestbook-1.0.0
-（中间省略）-
Creating tar archive
removing 'guestbook-1.0.0' (and everything under it)

$ ls dist/
guestbook-1.0.0.tar.gz
```

这样，我们就在 dist 目录下生成了 guestbook-1.0.0.tar.gz。这个 tar.gz 文件中包含 guestbook/__init__.py、setup.py、LICENSE.txt、HTML、CSS 等文件。

现在只要将这个文件放到我们想安装应用的环境中，就可以运行 `pip install guestbook-1.0.0.tar.gz`，直接从文件进行安装了。

3.3.6　提交至版本库

我们先将前面的内容提交到版本库。关于 hg 命令的操作，第 1 章和第 6 章中有详细介绍。目前的目录结构如 LIST 3.48 所示。

LIST 3.48　留言板项目的目录结构

```
/home/bpbook/guestbook/
   +-- .venv/
   +-- LICENSE.txt
```

```
    +-- MANIFEST.in
    +-- guestbook
    |    +-- __init__.py
    |    +-- static/main.css
    |    +-- templates/index.html
    +-- guestbook.dat
    +-- guestbook.egg-info/
    +-- setup.py
```

开发 Python 项目时，我们习惯将 setup.py 放在版本库最初级目录（根目录）下。这样我们就能用 pip 直接从版本库进行安装了。

另外，有些文件和目录是不用保存到版本库中的。guestbook.dat 文件的作用是记录留言板接收到的数据，这些数据没必要记录到版本库里。

guestbook.egg-info 目录的作用是记录程序包的元数据。元数据将在执行 `pip install -e .` 时自动生成。如果缺乏元数据，editable 安装可能无法正常进行。不过，由于它是在安装时自动生成的，所以也不用保存到版本库。

“.venv”也可以重新生成，因此不必保存到版本库。

接下来，我们需要将除上述三者以外的文件提交给版本库（LIST 3.49）。

LIST 3.49　注册到版本库

```
$ cd ~/guestbook
$ hg init
$ hg add LICENSE.txt MANIFEST.in guestbook setup.py
$ hg ci -m "initial"
```

另外，如果在目前的状态下执行 `hg status` 命令，刚才那些不需要上传的文件和目录仍会显示为非管理对象文件。我们需要在“.hgignore”文件中添加设置，将这些不需要管理的文件剔除出显示对象（LIST 3.50）。

LIST 3.50　.hgignore

```
.*.egg-info
^guestbook.dat$
^.venv$
```

“.hgignore”文件也要提交上去（LIST 3.51）。这样一来，在其他环境中使用 clone 的版本库时也就不会显示这些文件了。

LIST 3.51　提交“.hgignore”

```
$ hg add .hgignore
$ hg ci -m "add ignore list"
```

我们先暂且将其 push 到版本库服务器上。各位请在 Bitbucket 上创建一个空的 guestbook 项目，然后执行 LIST 3.52 所示的命令。

LIST 3.52　hg push

```
$ hg push https://bitbucket.org/< 你的 Bitbucket 账户 >/guestbook
```

建议各位今后适时地将添加、修改过的源码提交到版本库中。

专栏　Bitbucket

Bitbucket 是 Mercurial 的版本库服务器。Bitbucket 为用户提供了许多免费功能，可以管理 Mercurial 和 Git 版本库。

各位请先注册账户，创建空的 Mercurial 版本库，然后执行 LIST 3.53 所示的命令，完成 push 操作（本例中的用户名为 beproud，版本库名为 guestbook）。

LIST 3.53　hg push 示例

```
$ hg push https://bitbucket.org/beproud/guestbook
```

3.3.7　README.rst——开发环境设置流程

下面我们来描述设置流程说明书，总结该留言板应用开发环境的搭建流程。我们前面讲到的流程如下。

① clone 项目的版本库

② 搭建项目专用的 virtualenv 环境

③ 在 virtualenv 环境内执行 `pip install <directory>`（如果用于开发，则执行 `pip install -e <directory>`）

LIST 3.54　设置流程

```
$ hg clone https://bitbucket.org/beproud/guestbook
$ cd guestbook
$ virtualenv .venv
$ source .venv/bin/activate
(.venv)$ pip install .
(.venv)$ guestbook
 * Running on http://127.0.0.1:5000/
```

我们来把 LIST 3.54 中的流程原封不动地写入 README.rst。扩展名为 .rst 的文件是用

reStructuredText（reST）语法描述的文本文件。一般说来，Python 项目都会选用 reST 语法来写 README.rst。关于 reST 语法，我们将在第 7 章中详细了解。

通常，README.rst 包含 LIST 3.55 所示内容即可。

LIST 3.55　README.rst

```
===================
留言板应用
===================

目的
=====

练习开发通过 Web 浏览器提交留言的 Web 应用程序

工具版本
====================

:Python:     2.7.8
:pip:        1.5.6
:virtualenv: 1.11.6

安装与启动方法
=======================

从版本库获取代码，然后在该目录下搭建 virtualenv 环境 ::

   $ hg clone https://bitbucket.org/beproud/guestbook
   $ cd guestbook
   $ virtualenv .venv
   $ source .venv/bin/activate
   (.venv)$ pip install .
   (.venv)$ guestbook
    * Running on http://127.0.0.1:5000/

开发流程
=========

用于开发的安装
------------------

1. 检测
2. 按以下流程安装
```

```
(.venv)$ pip install -e .
```

写完之后要记得将 README.rst 文件提交到版本库。

各位请注意，我们在安装流程中写的是直接安装，但在开发流程中写的却是用 `pip install -e` 进行安装。而关于这二者的区别，我们并没有在 README.rst 文件中提及。这是因为我们认为阅读这篇文档的人应该懂得如何使用 pip 的 -e 选项，知道有它和没它的不同。使用普及率较高的工具或选项的优势就在于此。即便我们阅读文档时不知道它是什么，也能立刻查到相关资料，或者根据类似知识进行摸索。

专栏　缩短、定型化环境的搭建流程

时间一久，就算是自己开发的项目，我们也会忘记如何搭建运行环境。所以为了将来不忘记，我们最好在文档的开头就记下运行程序之前所需的全部流程。另外，尽量能让自己在看文档时立刻回想起当时用了什么流程。因此，流程要尽量短，而且要用开发者们普遍采用的结构。

以常用命令定型化的简洁流程具有以下优势。

- 不容易出现键入错误、流程颠倒等人为失误
- 减少整个项目中需要记忆的东西
- 需要向其他开发者或使用者传递的信息更少，减少文档量
- 测试和部署更容易自动化

3.3.8　变更依赖包

留言板的依赖包是 Flask。但是，我们很难在开发初期就确定好一款应用程序内的所有依赖包，有些时候还会放弃当前的包而改用其他的。特别是周期短、发布频繁的项目，往往每发布一次都会变更一次依赖包。

举个例子，假设我们放弃 Flask 改用 Bottle。这时如果直接用 pip 命令安装了 Flask 或 Bottle，那就必须将这一步骤告知其他开发者甚至是未来的自己（LIST 3.56）。

LIST 3.56　用 pip 替换了程序包，这一步该如何告知其他人

```
(.venv)$ pip uninstall flask
(.venv)$ pip install bottle
```

留言板的 setup.py 里记录着依赖包的信息，因此我们只需更改 setup.py 的设置即可。如果改写了 setup.py 的 install_requires 行，需要再次执行 `pip install -e .`。

这一步骤的命令和安装时的命令一样，因此不需要修改流程说明书。只要其他新建项目环

境的开发者执行了 `pip install -e .` 命令，就能安装好该项目所需的全部程序包。

不过，还有一点需要注意。那就是，即使我们从 setup.py 中删除了 flask，之前安装到环境中的 Flask 及其关联程序包也不会被卸载。要想删除已经无用的程序包，需要用 `virtualenv --clear .` 等方法重建环境（LIST 3.57）。

LIST 3.57　重建环境

```
(.venv)$ virtualenv --clear .venv  # 删除 .venv 环境内的全部依赖库
(.venv)$ pip install -e .          # 根据 ./setup.py 安装依赖库
```

这一处理会重新安装依赖库，所以运行时将占用较长时间。

> **NOTE**
>
> 建议各位设置 pip 的 --download-cache 选项，缩短下载时间（pip-6.0 之前的版本）。使用第 9 章中介绍的 wheelhouse 能进一步加快速度。

> **NOTE**
>
> 关于如何固定开发环境中安装的程序包的版本，各位请参考第 9 章。另外，第 9 章还会讲解强制指定依赖包范围的相关知识。

另外，最好在 README.rst 中添加 LIST 3.58 所示的流程。

LIST 3.58　README.rst

```
开发流程
=========

变更依赖库时
---------------------

1. 更新 ``setup.py`` 的 ``install_requires``
2. 按以下流程更新环境 ::

     (.venv)$ virtualenv --clear .venv
     (.venv)$ pip install -e ./guestbook

3. 将 setup.py 提交到版本库
```

3.3.9　通过 requirements.txt 固定开发版本

前面我们介绍了用 setup.py 管理依赖包的方法。实际上，我们还可以用 requirements.txt 管理依赖库。

setup.py 是在 PyPI 上发布程序包时必不可少的组成部分，而且安装时的依赖库也需要用 setup.py 的 install_requires 来指定。这种情况下，由于使用该包的各个环境大不相同，所以我们不能严格指定依赖库的版本，只能指定最低需求。当然，我们还可以手动编辑 requirements.txt，免除指定版本的工作。但要知道，pip install guestbook 是不会引用 requirements.txt 的，就算我们将 requirements.txt 与发布的程序包捆绑在了一起，计算机仍然不会自动安装依赖库。

相反，如果我们的项目不需要封装，只是被拿来当作一个 Web 应用在服务器上发布，就没必要使用 setup.py 了。在项目从开发到正式上线的过程中，有许多程序库和应用程序要一个版本用到底，因此我们要严格地指定版本。而对于不需要发布也不需要封装的项目，setup.py 就失去了用处。对这些项目而言，用 requirements.txt 效率更高。

创建 requirements.txt 的命令如 LIST 3.59 所示。

LIST 3.59　创建 requirements.txt

```
(.venv)$ pip freeze > requirements.txt
```

requirements.txt 中记载着当前环境内已安装的所有程序包及明确的版本号（LIST 3.60）。

LIST 3.60　requirements.txt

```
Flask==0.10.1
Jinja2==2.7.3
MarkupSafe==0.23
Werkzeug==0.9.6
guestbook==1.0.0
itsdangerous==0.24
```

用 setup.py 管理依赖包时，我们只写了 Flask 但没有指定版本。这是 setup.py 管理和 requirements.txt 管理的一大区别。

要想在其他环境安装同样的程序包们，我们需要将这个 requirements.txt 文件放到该环境下，然后用如 LIST 3.61 所示的方法安装。

LIST 3.61　用 requirements.txt 进行安装

```
(.venv)$ pip install -r requirements.txt
```

这样一来，环境中就安装了同样版本的程序包。

现在将创建好的 requirements.txt 文件也提交到版本库。另外，当我们变更依赖包时要记得更新这个文件。请各位打开 README.rst 文件，将变更依赖库时的流程更新成如 LIST 3.62 所示的内容。

LIST 3.62 README.rst

```
开发流程
=========

变更依赖库时
---------------------

1. 更新 ``setup.py`` 的 ``install_requires``
2. 按以下流程更新环境 ::

     (.venv)$ virtualenv --clear .venv
     (.venv)$ pip install -e ./guestbook
     (.venv)$ pip freeze > requirements.txt

3. 将 setup.py 和 requirements.txt 提交到版本库
```

在这个流程中，依赖包同时被 setup.py 和 requirements.txt 两个文件管理着。至于该用 setup.py 管理还是 requirements.txt 管理，要视项目的公开方式或使用方式而定。

3.3.10 python setup.py bdist_wheel——制作用于 wheel 发布的程序包

接下来，我们制作 wheel 程序包。wheel 程序包的使用方法在 9.1 节有详细讲解。

制作 wheel 程序包之前，我们先安装 wheel（LIST 3.63）。

LIST 3.63 安装 wheel

```
$ pip install wheel
Downloading/unpacking wheel
  Downloading wheel-0.24.0-py2.py3-none-any.whl (63kB): 63kB downloaded
  Installing collected packages: wheel
  Successfully installed wheel
  Cleaning up...
```

安装完成之后，我们就可以用 `bdist_wheel` 命令了。接下来，我们执行 `python setup.py bdist_wheel` 来生成 wheel 程序包（LIST 3.64）。

LIST 3.64　生成 wheel 程序包

```
$ python setup.py bdist_wheel
running bdist_wheel
-(中间省略)-
creating _build/bdist.linux-x86_64/wheel/guestbook-1.0.0.dist-info/WHEEL

$ ls dist/
guestbook-1.0.0-py2-none-any.whl   guestbook-1.0.0.tar.gz
```

执行完之后，dist 目录下就会生成 guestbook-1.0.0-p2-none-any.whl。这个扩展名为“.whl”的文件就是 wheel 程序包。在 wheel 程序包内，按照安装后的目录结构捆绑了源码和各种文件。与源码程序包不同，它里面没有 setup.py。

现在只要将这个文件复制到等待安装的环境中，我们就可以执行 `pip install guestbook-1.0.0-p2-none-any.whl`，直接从文件进行安装了。由于这时不需要运行 setup.py，所以会比源码程序包的安装速度快出一大截。

专栏　Universal Wheel：同时支持 Python2 和 Python3 的 wheel 程序包

我们将 Python2 系列和 Python3 系列都可以用的 wheel 程序包称为 Universal Wheel。

刚才我们生成的程序包是 guestbook-1.0.0-p2-none-any.whl，从名字上就可以看出，这个程序包是对应 Python2 的。由于 guestbook 同时支持 Python3，所以我们在 Python3 下执行上述流程时，会生成名为 guestbook-1.0.0-p3-none-any.whl 的 wheel 文件。

如果想生成 Universal Wheel 程序包，就需要在 bdist_wheel 后面加上 --universal 选项，代码如下。

```
$ python setup.py bdist_wheel --universal
--(中间省略)--
$ ls dist
guestbook-1.0.0-py2.py3-none-any.whl
```

当程序包仅由纯 Python 代码实现时，会生成 Universal Wheel。而在诸如需要二进制构建、Python 实现方面受限等情况下，则无法生成 Universal Wheel。

3.3.11　上传到 PyPI 并公开

我们之所以能用 pip 命令安装指定的程序包，是因为这些包都被注册到了 PyPI 上。PyPI 是 Python 的官方网站，所有人都能随意上传及下载 Python 程序包。如果各位不介意公开自己开发的程序包，不妨将它注册到 PyPI 上。

举个例子，如果我们要安装一个已经在 PyPI 上注册的程序包 bpmappers，那么只需执行 `pip install bpmappers` 即可。

PyPI 的作用相当于一台负责分发程序包的中央服务器。不过，注册到 PyPI 的程序包无法只对特定用户公开，所以各位要千万注意，别把对外保密的程序库注册上来。

NOTE

在这种情况下，我们可以在公司内部准备一台 PyPI 交换服务器，或者找其他等价的方法。这类方法我们将在第 9 章中详细介绍。

下面我们就把已做好的程序包文件注册到 PyPI。如果想在实际注册之前先注册到测试服务器，可以参考本节的专栏“PyPI 的测试服务器”。

执行 `register` 命令，注册 guestbook 程序包（LIST 3.65）。

LIST 3.65 注册程序包

```
$ python setup.py register
```

如果发生下述情况，执行 register 命令时会被询问是否拥有 PyPI 账户。

- 该环境第一次执行 register 命令
- 保存账户信息的 .pypirc 文件无效

发生上述情况时，我们会接到如 LIST 3.66 所示的账户询问信息。

LIST 3.66 注册程序包时的账户询问

```
$ python setup.py register
running register
...
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to you), or
4. quit
Your selection [default 1]:
```

如果各位已经有 PyPI 账户，请选 1 并输入用户名和密码。如果没有，则需要先去 PyPI 网站创建账户之后再选 1，要么就是直接选择 2 或 3。这步操作会认证我们的 PyPI 账户，只要认证成功，我们就可以使用 `register` 和 `upload` 命令操作 PyPI 了。

专栏 在 .pypirc 上保存密码时的注意事项

在输入完用户信息、即将完成注册时，系统会询问是否将登录信息保存在主目录的“.pypirc”文件中。如果输入 Y，计算机会自动生成“.pypirc”文件，以纯文本形式保存用户名和密码。因此，我们最好采取一些对策（比如设置权限等），防止“.pypirc”文件的内容被第三者窃取。

另外，从 Python 2.7 开始，我们可以用编辑器将保存后的“.pypirc”文件的 password 栏设置为空栏。此后再进行上传时，只要用 `register upload` 命令代替单独的 `upload` 命令，就可以做到仅执行时验证密码。

完成注册之后，就可以向 PyPI 上传程序包了。执行下述命令之后，源码程序包就会被上传至 PyPI。

```
$ python setup.py sdist bdist_wheel upload
```

刚才我们单独执行了 `sdist` 命令和 `bdist_wheel` 命令。其实如上例所示，只要在命令末尾指定 `upload` 命令，就可以在封装 sdist 和 bdist_wheel 程序包之后直接将它们设为上传对象。

绝大部分情况下，一个项目每次发布的程序包类型都基本一致。因此，我们可以把注册新版本、构建程序包、上传这一系列流程整合成一个命令。整合多条命令时，需要用到 alias 功能，代码如下。

```
$ python setup.py alias release register sdist bdist_wheel upload
$ python setup.py release
```

`alias` 命令会把设置保存在 setup.cfg 中，所以我们要把这个文件也提交到版本库里。共享 alias 可以一定程度上避免项目其他成员在发布时出现失误。

专栏 PyPI 的测试服务器

如果严格执行本书中的流程，各位的 guestbook 程序包就会被实际注册到 PyPI 上。但这毕竟是一个练习，我们不建议各位向 PyPI 服务器注册这些练习性质的东西（请在 PyPI 网站搜索 printer）。

所以，请各位在练习时使用 PyPI 的测试服务器 TestPyPI[①]。TestPyPI 是一个对所有人开放的服务器，专门用来供人们做实验。我们可以用它来练习向 PyPI 上传程序包以及从 PyPI 下载程序包。

具体使用方法请参考 TestPyPI 的说明页面[②]。

① https://testpypi.python.org/pypi
② https://wiki.python.org/moin/TestPyPI

◉ 描述程序包的详细信息

上传完成后，PyPI 会给该程序包分配一个 URL。例如 guestbook 程序包的 URL 是 `https://pypi.python.org/pypi/guestbook`。另外，setup.py 的 long_description 传值参数所指定的内容将会显示在 PyPI 页面上。

以刚才编写完成的 setup.py 为例来看，我们会发现 PyPI 页面上只显示了下载文件的列表，并没有任何详细说明。这是因为我们并没有指定 long_description。要知道，除了下载列表之外，还有很多对使用者有帮助的信息，所以我们应该将这些信息描述在 setup.py 中，让人们能在 PyPI 上看到它们。下面是一个描述示例。

```
import os
from setuptools import setup, find_packages

def read_file(filename):
    basepath = os.path.dirname(os.path.dirname(__file__))
    filepath = os.path.join(basepath, filename)
    if os.path.exists(filepath):
        return open(filepath).read()
    else:
        return ''

setup(
    name='guestbook',
    version='1.0.0',
    description='A guestbook web application.',
    long_description=read_file('README.rst'),
    author='< 你的名字 >',
    author_email='< 你的邮箱地址 >',
    url='https://bitbucket.org/< 你的 Bitbucket 账户 >/guestbook/',
    classifiers=[
        'Development Status :: 4 - Beta',
        'Framework :: Flask',
        'License :: OSI Approved :: BSD License',
        'Programming Language :: Python',
        'Programming Language :: Python :: 2.7',
    ],
    packages=find_packages(),
    include_package_data=True,
    keywords=['web', 'guestbook'],
    license='BSD License',
    install_requires=[
        'Flask',
```

```
    ],
    entry_points="""
        [console_scripts]
        guestbook = guestbook:main
    """,
)
```

这里添加的项目有以下 4 个。

- long_description

  可描述多行说明。说明内容按照reStructuredText(reST)语法进行描述，PyPI会将其自动转换为HTML并显示在网站上。long_description的内容大多和README.rst相同。即便不同，我们也建议各位将长达几行的说明文章存放在其他文件中。在上面的例子里，我们设置了让long_description读取README.rst文件。

- classifiers

  从trove classifiers定义的项目中选取适当项目，以列表的形式列举在这里。这个列表包含的项目就是程序包在PyPI上的分类。用户可以在PyPI网站上通过分类筛选来寻找自己想要的程序包。

  在上面的例子里，我们描述了许可证信息和Python版本等。如果我们想指定的分类并不在trove classifiers之中， 那么指不指定它都无所谓。

- keywords

  以列表形式列举出易于搜索的单词，或者让使用者一眼就明白意思的词汇。

- license

  可随意描述许可证信息。前面我们只能用trove classifiers定义过的值来指定classifiers，但license处却可以指定任意字符串。在上例中，我们将这部分描述为BSD License。

这里只对一部分会显示在 PyPI 上的 setup 函数的传值参数进行了介绍，此外还有许多这里并未提及的传值参数。

> **Python 官方文档**
>
> https://docs.python.org/2.7/distutils/setupscript.html#additional-meta-data

◉ 检查 setup 函数内指定的参数

将程序包实际上传到 PyPI 之后，我们需要打开 PyPI 页面查看一下效果。如果发现页面并没有将 long_description 的内容转换为 HTML，而是直接显示了 reST 文本，那么很可能是我们的 reST 描述出现了错误。为回避这一问题，最好在 upload 之前查一遍错。查错时可以用

docutils 附属的 rst2html.py 命令。docutils 是一个文字处理工具，它能将 reST 文本转换成其他多种格式。其安装方法如下。

```
$ pip install docutils
```

安装好 docutils 后，用下述方法执行 rst2html.py 命令，查看文档内是否有错误描述。

```
$ python setup.py --long-description | rst2html.py > /dev/null
```

另外，如果各位使用的是 Python 2.7 以上的版本，并且环境中安装了 docutils，那么可以用下述简短的命令来查错。文档没有问题的情况下只会显示 running check，有问题则会显示类似下例的内容。

```
$ python setup.py check -r -s
running check
warning: check: No directive entry for "spam" in module "docutils.parsers.rst.
languages.en".
Trying "spam" as canonical directive name. (line 19)

warning: check: Could not finish the parsing.

error: Please correct your package.
```

check 命令用于检查 setup 函数各参数的设置是否正确。如果参数遗漏或指定有误，系统会报错或者发出警告。我们只要给 check 命令指定几个选项，就可以检查 long_description 中指定的文本是否符合 reStructuredText 语法了。

这个检测命令也可以添加到我们前面提过的 alias 设置当中，具体如下例所示。

```
$ python setup.py alias release check -r -s register sdist bdist_egg upload
```

这样一来，在我们执行 python setup.py release 命令时，系统就会先进行 check。一旦 check 发现问题，后续的 register 命令将不再执行，有问题的程序包也就不会被发布出去了。

专栏 在 PyPI 上公开

不知各位有没有这样一种感觉：在 PyPI 上公开了程序包的 Python 工程师都是大牛！

“写好程序之后，先整理成 Python 的标准发布形式，然后公开到 PyPI 上，让全世界人拿去用！”这话说起来容易，但真正做起来时，许多人会不禁感到犹豫。原因主要有两个，一来是不知道怎么生成标准的发布包，二来是认为自己写的程序别人拿去也没什么用。当然可能还有心理上的抗拒，觉得 PyPI 上面满满的都是世界顶尖好用的程序，自己写的程序难登大雅之堂，没

资格发布在 PyPI 上。

那么，我们回过头来想一下，在 PyPI 上公开程序包的最大动力又是什么呢？这个问题显然因人而异，但绝大多数人的出发点无外乎"希望得到程序的反馈信息""希望能帮到别人""向 Python 工程师同行们炫耀一下""想听到别人的夸赞"之类。另外，我们平时也会隔三差五地向 PyPI 上传几个程序包，而一直支持我们的动机则是"想为组织或企业做宣传""想让大家知道，让大家拿去用""很多自己平时用的 OSS 都来自这里，所以想把自己做出的成果也放在这上面，为 OSS 界做一份贡献"。

在公开程序包时，"保证品质""整理文档""进行测试"这三项工作必不可少。这里，"不想让全世界的工程师看自己出丑"的心理因素只是一小方面，更大的原因是这个阶段能完善我们的程序库，使其回到一个干净的状态，避免在日常开发的过程中混入一些多余功能。当然，公开后获得的反馈也是其魅力之一。

不但个人编写的程序如此，工作中开发出来的程序同样是这个道理。拿我们 BePROUD 来说，bpmappers 和 bpssl 就是例子。因为它们向一般公众公开，所以要维持适当的功能和文档。我们同时还收到了公司内外两方面的反馈，这让我们能不断作出改进。

所以，让我们都来做在 PyPI 上公开了程序包的 Python 工程师大牛吧。这对技术上的要求并没有各位想象中那么高。

3.3.12　小结

发布采用 Python 编写的程序库或应用程序时，最标准的方法就是用 setup.py 进行封装。经 setup.py 封装后的发布包要注册到 PyPI 上，然后其他用户就可以通过 pip 轻松地安装了。另外，对于一些不打算公开的程序，我们也建议将其封装成可发布的状态，这样既能方便地放到其他环境下试运行，又能便于其他项目拿去重复利用。此外还要养成一个习惯，即在 README.rst 文件中写明项目概要、运行方法、设置等信息，以便重复利用程序包。

如果各位还想进一步了解 setup.py 的写法或封装的相关知识，可以参考 Python Packaging User Guide。

Python Packaging User Guide
`https://packaging.python.org/en/latest/`

3.4 小结

本章我们学习了 Python 项目的结构，以及如何对使用 PyPA 工具的开发环境进行设置。另外，我们对第 2 章编写的程序进行了结构调整，同时学习了如何把程序包上传至 PyPI。

对于要经常重建环境的情况（例如自动测试或向服务器部署）而言，本章介绍的上传流程更能发挥其效果。另外，从下一章开始，我们的介绍重点将从个人开发转移到团队开发，而这里学习的流程对团队开发同样有效。

第 2 部分

团队开发的周期

在第 2 部分中，我们将学习团队开发中必备的思路，以及一些支持团队开发的技术和工具。本部分以多人同时作业为前提，内容涉及多人协同开发所必须的堆栈及源码共享、文档的编写、单元测试、封装、持续集成等方面。

第 4 章　面向团队开发的工具

前面我们用 3 章的篇幅学习了个人如何搭建开发环境以及开发应用。独立开发的时候，保存源码等工作完全可以按照自己喜欢的方法来，而且各种信息和点子直接写到自己的笔记本上就好。

那么，一个由多人组成的团队应该如何进行开发呢？首先，开发出来的成品需要在所有成员间共享。然后，还要有一个用来测试开发成品的环境。另外，需求和技术等信息也必须共享给所有成员，而且成员间交流的场所也必不可少。

本章将以提高团队开发效率为目的，为各位介绍团队开发环境的搭建过程以及一些有用的工具。

4.1　问题跟踪系统

要想让团队开发顺风顺水，首先就要把握每个开发任务的负责人是谁、开发进展到了什么程度等信息，并把这些信息整理起来。在这类任务管理工作上，问题跟踪系统（Issue Tracking System，ITS）是一把好手。

我们可以将开发中的任务添加到问题跟踪系统，然后用该系统来追踪、管理这些任务的状态。多数问题跟踪系统会以问题（Ticket）为单位进行项目管理。一般情况下，我们会把工作任务以问题的形式添加到问题跟踪系统，而不是单纯拿它来管理项目。

问题跟踪系统可以给问题添加状态（新建、进行中、解决、结束、驳回等），设置优先级（紧急、高、普通、低）、负责人、日期等，并且允许用户通过搜索查看信息一览，让我们能随时把握项目的开发情况。

将作业分割成一个个任务，再将任务分配给问题进行管理，这种开发手法称为问题驱动开发（Ticket Driven Development，TiDD）。它与敏捷开发有着很好的相容性，因此在近年来的开发中人们逐渐开始实践这种手法。关于问题跟踪系统的实际运用以及问题驱动开发的相关知识，我们将在第 5 章中进行学习。

4.1.1　Redmine

接下来，我们来了解一款问题跟踪系统——Redmine[①]。Redmine 是一款开源的问题跟踪系

① http://www.redmine.org/

统，它以管理项目内的任务和 Bug 的问题功能为中心，兼具服务于团队开发的功能，比如与 Wiki 和版本控制系统的联动等。

4.1.2　安装 Redmine

现在我们来安装 Redmine。首先要安装的是 Web 服务器（Apache）和 Passenger（运行 Rails 应用所需的 Apache 模块）（LIST 4.1）。

LIST 4.1　安装 Apache

```
$ sudo apt-get install -y apache2 libapache2-mod-passenger
```

然后安装数据库 MySQL（LIST 4.2）。

LIST 4.2　安装 MySQL

```
$ sudo apt-get install -y mysql-server mysql-client
```

在安装 MySQL 的过程中，需要设置 MySQL 的 root 用户密码（图 4.1、图 4.2）。这个密码在随后安装 Redmine 时会用到。



图 4.1　MySQL 的密码设置



图 4.2　MySQL 的密码设置（确认）

接下来安装 Redmine（LIST 4.3）。

LIST 4.3　安装 Redmine

```
$ sudo apt-get install -y redmine redmine-mysql
```

安装过程中会显示数据库设置的相关界面，请选择“是”(图 4.3)。

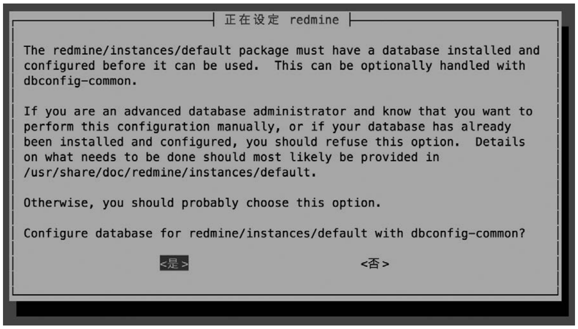


图 4.3　数据库设置

下一步选择我们要使用的数据库。这里选刚才安装的 MySQL(图 4.4)。



图 4.4　选择数据库

输入 MySQL 数据库管理员的密码。这里输入刚刚安装 MySQL 时设置的密码 (图 4.5)。



图 4.5　管理员密码输入界面

最后设置 Redmine 专用的 MySQL 应用密码。设置加上确认，密码总共要输入两遍 (图 4.6、图 4.7)。



图 4.6　设置 MySQL 应用密码



图 4.7　确认 MySQL 应用密码

至此，Redmine 安装完毕。

4.1.3　Redmine 的设置

要想通过 Web 浏览器访问 Redmine，需要更改 Apache 的设置，所以接下来我们修改 Apache 的设置文件。LIST 4.4 中的设置将这个 Web 服务器设置成了 Redmine 专用。

LIST 4.4　修改 Apache 的设置文件

```
$ sudo vi /etc/apache2/sites-available/000-default.conf
```

替换 `000-default.conf` 的如下部分（LIST 4.5）。

LIST 4.5　000-default.conf 的修改

```
DocumentRoot /var/www/html
↓
DocumentRoot /usr/share/redmine/public
```

Redmine 需要一个名为 bundler 的工具，以管理 Gem 包，所以我们还需要安装 bundler（LIST 4.6）。

LIST 4.6　安装 bundler

```
$ sudo gem install bundler --no-rdoc --no-ri
```

为保证我们能通过 Apache 对 redmine 目录下执行写入操作，需要修改文件的所有权

（LIST 4.7）。

LIST 4.7　变更文件的所有权

```
$ sudo chown -R www-data:www-data /usr/share/redmine
```

至此，所有设置结束，我们重新启动 Apache（LIST 4.8）。

LIST 4.8　重启 Apache

```
$ sudo service apache2 restart
```

如果要把 Redmine 安装在客 OS 上，然后通过本地环境（主 OS）的 Web 浏览器浏览界面，则必须设置端口转发。方法与附录 B.2 一样，需要在 VirtualBox 的端口转发设置中添加一行设置：主机端口（TCP）8000、子系统端口 80（图 4.8）。

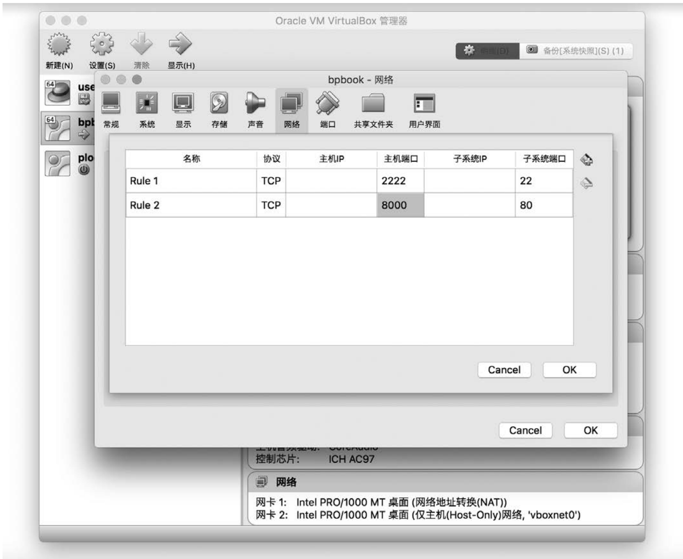


图 4.8　VirtualBox 端口转发设置

现在各位可以打开浏览器访问 `http://127.0.0.1:8000/`，看看是否能显示出 Redmine 的界面。如果全部设置正确，各位应该会看到如图 4.9 所示界面。

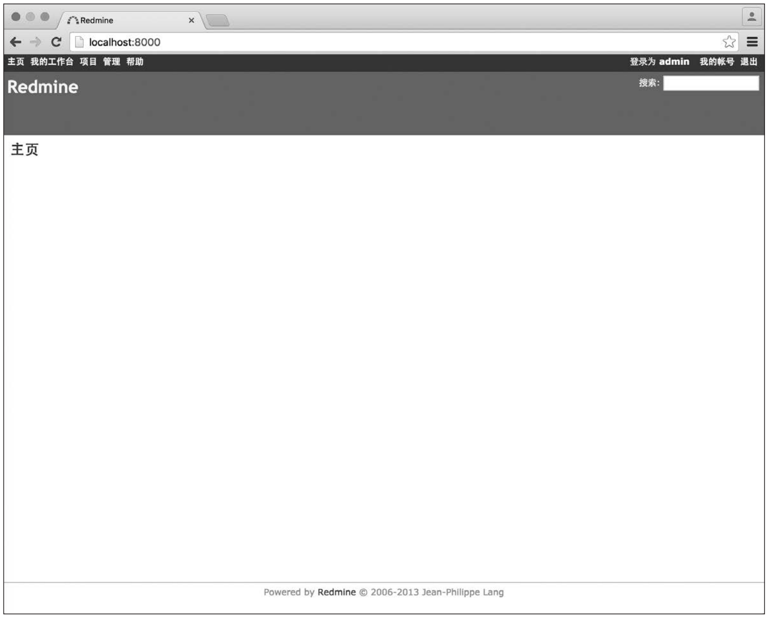


图 4.9　Redmine 的初始界面

怎么样，看到 Redmine 的界面了吗？

在初始状态下，管理员只要在用户名和密码处都填入 admin 即可登录。请各位登录管理员用户，修改 Redmine 的各项设置。

4.1.4　插件

Redmine 有多种用来扩展功能的插件。下面我们挑几个比较好用的来了解一下。

◉ Redmine reStructuredText Formatter

Redmine reStructuredText Formatter[①] 用来将问题和 Wiki 的语法从标准 Textile 转换为 reStructuredText（reST）。

具体使用哪种语法，我们可以按照团队的喜好来定。用 reST 描述的好处在于方便用 Sphinx 整理文章。我们将在第 7 章中学习 Sphinx 的相关内容。

◉ SCM Creator plugin

使用 SCM Creator plugin[②] 之后，我们就可以通过 Redmine 给项目创建版本库了。

① https://github.com/ebrahim/redmine_restructuredtext_formatter

② http://www.redmine.org/plugins/redmine_scm

这个插件的安装和设置请参考 4.2 节。

◉ Slack chat plugin

导入 Slack chat plugin[①] 之后，Slack 聊天系统将能够接收 Redmine 中的问题更新通知。关于 Slack，我们将在 4.3 节进行了解。

◉ Issue Template plugin

使用 Issue Template plugin[②] 可以为每种问题分别设置模板，以便开发团队成员明白哪种问题该描述些什么。比如，我们给 Bug 问题设置了模板后，就可以有效防止漏记项目。

Issue Template plugin 的安装和使用方法请参考 5.2 节。

4.2　版本控制系统

在进行团队开发时，我们需要一个地方来集中管理和共享各成员的开发成果。另外，在团队开发的过程中，程序内难免混入 Bug，这就需要对开发成果进行历史管理，以便追踪并掌握 Bug 混入的时间点。

在这一点上，版本控制系统（Version Control System，VCS）能提供很大帮助。顾名思义，版本控制系统就是用来管理源码等内容的开发历史的。当今的版本控制系统主要有集中式的 Subversion 和分布式的 Mercurial、Git。本书选用的是 Mercurial。

Redmine 和 Mercurial 都是面向团队开发的工具，这里将学习如何把它们结合在一起使用。Mercurial 的安装与简单的用法请参考第 1 章，实用性用法请参考第 6 章。

4.2.1　Mercurial 与 Redmine 的联动

如果让版本控制系统和问题跟踪系统两个系统联动起来，可以明确源码的变更和问题之间的对应关系，让二者相得益彰。现在我们就来了解一下 Redmine 和 Mercurial 的联动方法。

如果各位的服务器上还没有安装 Mercurial，那么需要先安装 Mercurial，然后重启 Redmine（LIST 4.9）。

LIST 4.9　安装 Mercurial

```
$ sudo pip install mercurial
$ sudo service apache2 restart
```

---

① https://github.com/sciyoshi/redmine-slack

② http://www.r-labs.org/projects/issue-template/

以管理员身份登录 Redmine，设置版本控制系统。

在图 4.10 所示的配置界面选择“版本库”标签页，勾选“启用 SCM”中的相应系统然后保存。这样我们就可以使用选中的版本控制系统了。



图 4.10　Redmine 的版本库配置界面

4.2.2　用于生成版本库的插件

虽然实现版本控制系统的联动就可以了，但是我们还希望能直接从 Redmine 创建版本库，这样会更加方便。为此，我们还要安装 SCM Creator plugin① 并进行设置。

首先创建一个用于生成版本库的目录（LIST 4.10）。这里我们用的目录为 /var/lib/hg，然后更改它的所有权，让我们能从 Redmine 对它进行写入操作。

LIST 4.10　创建用于版本库的目录

```
$ sudo mkdir /var/lib/hg
$ sudo chown www-data:www-data /var/lib/hg
```

然后创建设置文件 /usr/share/redmine/config/scm.yml（LIST 4.11）。下例是最低限度的设置。关于设置项目的详细资料，请参考该插件的 Web 页面。

LIST 4.11　scm.yml

```
production:
  deny_delete: false
  auto_create: true
```

① http://projects.andriylesyuk.com/projects/scm-creator

```
force_repository: false
max_repos: 0
only_creator: true
allow_add_local: false
allow_pickup: false
mercurial:
  path: /var/lib/hg
  hg: /usr/local/bin/hg
```

插件的安装流程如 LIST 4.12。需要先将下载好的插件解压到 Redmine 的 plugins 目录下，然后执行 rake 命令构建。

LIST 4.12　安装 SCM Creator plugin

```
$ sudo mkdir /usr/share/redmine/plugins
$ wget http://projects.andriylesyuk.com/attachments/download/563/redmine_scm-
0.5.0b.tar.bz2
$ sudo tar xfj redmine_scm-0.5.0b.tar.bz2 -C /usr/share/redmine/plugins/
$ cd /usr/share/redmine/
$ sudo chown -R www-data:www-data plugins/redmine_scm/
$ sudo rake redmine:plugins:migrate RAILS_ENV=production
$ sudo service apache2 restart
```

重启 apache2 后，插件将被自动加载。我们可以在图 4.11 所示的配置界面里看到插件正常加载的信息。

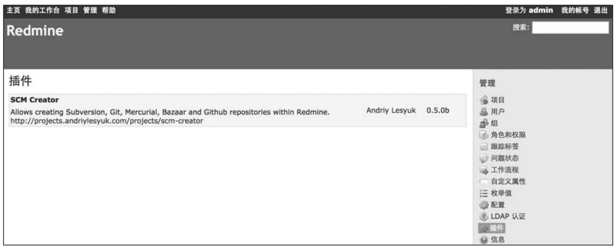


图 4.11　Redmine 的插件配置界面

接下来新建一个项目。选择界面左上方的“项目”，然后点击“新建项目”进入新建项目界面（图 4.12）。要记得先指定 SCM（这里指定为 Mercurial）再创建项目。



图 4.12　新建项目界面

查看刚刚创建好的项目的配置界面，可以看到“版本库”标签页中已经生成了 Mercurial 的版本库（图 4.13）。



图 4.13　版本库配置界面

至此，Mercurial 与 Redmine 的联动就实现了。

4.3　聊天系统

在团队开发中，必须保证报告、联络、探讨这些交流能顺利进行。说到交流，一般不是面对面开会就是邮件往来，然而面对面开会时，其内容很难与不在场的成员共享，而邮件又难以保证效率。聊天系统则能扬长避短，轻松实现远程实时对话以及团队内信息共享。

聊天系统种类繁多，比如 Skype、Slack、LINE、IRC 等。这里我们来了解一下 Slack[①]。

4.3.1　Slack

Slack 是一款面向企业和团体的交流工具。除了可以进行一对一的对话之外，还能创建多个

① https://slack.com/

频道、内部群组等聊天室，让多名团队成员同时参与对话（图 4.14）。

这项服务在 2013 年才刚刚起步，但由于其导入简单、使用方便而广受人们青睐，如今已被许多企业和团体使用。

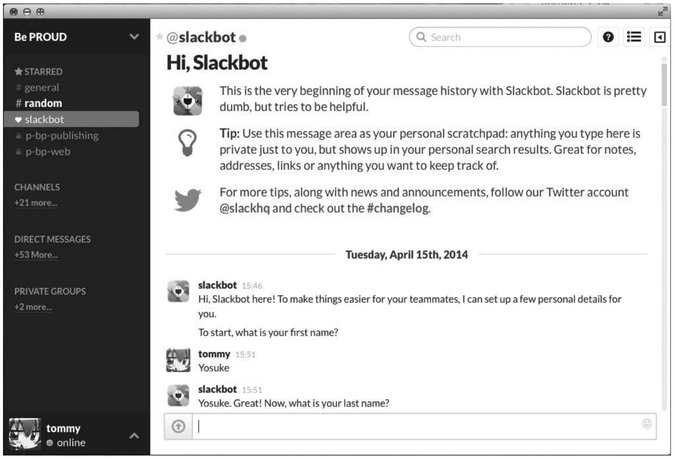


图 4.14　Slack

4.3.2　Slack 的特点

下面我们来了解一下 Slack 的主要特点。

◉ 能轻松导入的 Web 服务

虽然 Slack 有专用的客户端，但其很大程度上是基于 Web 的，所以能脱离客户端直接通过浏览器使用。它不像 IRC 那样需要准备服务器，再加上采用了免费增值模式，因此可以免费使用。导入和使用几乎零成本是它的巨大优势。

◉ 能与许多外部服务联动

Slack 可以实现与许多外部服务的联动（Integration）。它标配的 Integration 超过 60 种，其中包括了 Dropbox、GitHub、Google Drive、Twitter 等，我们可以从设置界面轻松地开启某项联动功能。联动的内容因服务而异，以 Google Drive 为例，如果我们在聊天室粘贴了 Google Drive 上的文档的 URL，聊天室内就会自动显示出文档的标题。

Redmine 与 Slack 的联动可由 Slack chat plugin for Redmine[①] 实现。这是一个要用到 Slack API 的插件，将它导入 Redmine 之后，Redmine 就能向 Slack 的聊天室发送通知了。

① https://github.com/sciyoshi/redmine-slack

除 Redmine 之外，Jenkins、Sentry 也可以向 Slack 发送通知（图 4.15）。Jenkins 直接用标配的 Integration 即可。如果使用 Sentry，则要给 Sentry 安装 sentry-slack 扩展[①]。

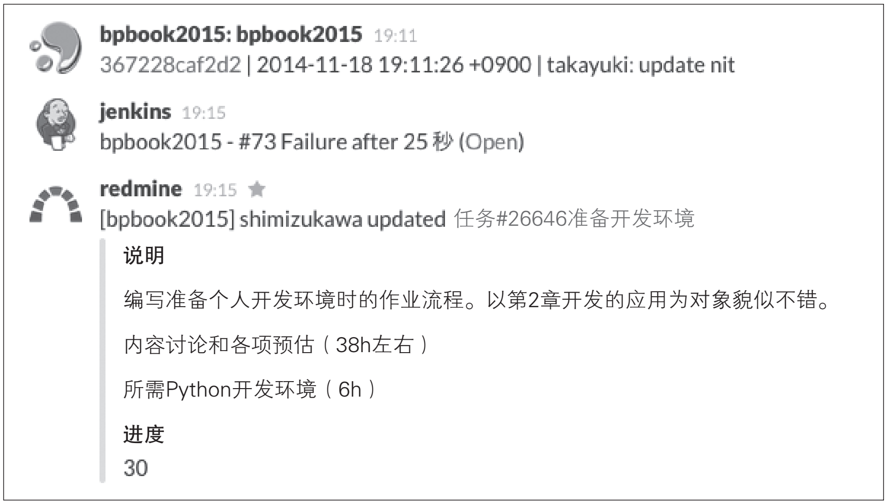


图 4.15　Slack 与 Redmine 和 Jenkins 的联动

通过实时显示问题的更新、向版本库的提交、Jenkins 的构建信息等，既可以方便我们掌握团队成员的工作情况，又可以保留作业日志，对团队开发而言是一种极大的帮助。

◉ 可创建 bot

Slackbot 是 Slack 标配的 bot。这个功能会对我们设置的关键字作出反应，然后返回特定的消息。比如，如果希望在有人说“我要回家了”的时候，自动回答“您辛苦了”这句话或者别的话，只要在 Slackbot 中进行相应设置就行了。

如果想进行更加复杂的处理，可以用 Slack 附带的 WebHook 功能来创建自己的 bot。

既然是自己的 bot，那它的用处就全看我们的创造力了。

举个例子，假设我们在 Google Drive 的电子表格里存了测试项目与测试结果的数据一览。现在每次查看测试进展情况都要打开一次电子表格，实在太麻烦，所以我们希望这一流程能实现自动化。这时就可以创建一个 bot，让它在电子表格更新时自动获取测试结果一览，并提交给 Slack。有了这个 bot 之后，团队所有成员都能共享测试的情况，这对团队开发大有助益。

◉ 强大的 @ 功能加速团队内的信息传递

Slack 有着强大的 @ 功能，这个功能可以让我们在发消息时指定呼叫某个用户。如果在我们使用 Slack 时，有人发消息并 @ 了我们，那么聊天室列表中会显示通知图标，告诉我们“有

① https://github.com/getsentry/sentry-slack

人 @ 你”。如果我们当时不在线，该消息内容会以邮件形式进行通知。另外，使用 iOS 或 Android 的 Slack 应用时，通知会显示在终端的通知区域里。由于它会借助各种通知手段来通知，因此我们很难漏掉那些 @ 自己的信息。此外，Slack 还有“@ 一览”功能，供用户查看以往所有 @ 自己的信息。

在 @ 别人时，用户名输入到一半会自动弹出用户名候选列表，我们只要从列表中选择用户名即可。这样一来既可以节省时间，又能防止输错用户名。如果要 @ 聊天室中的所有人，需要用 @channel、@group 等特殊名称。

正因为 Slack 的 UI 发 @ 方便、被 @ 醒目，所以很少会出现“想告诉某人一件事，结果某人没注意到”这种沟通失误。对使用 Slack 的团队开发而言，这绝对是一大优势。

◉ URL 和文件的预览功能

在聊天室贴 URL 或向聊天室上传文件后，成员们能预览其内容。Web 页面的 URL 会显示该页面内的文本，图片的 URL 则会显示图片。上传的文件也是同样道理，文本文件会显示文本，jpg 或 png 等图片文件则会显示图片。

借助于这一功能，我们能很轻松地与团队成员共享图片形式的信息（比如尚在开发中的设计方案）。

◉ 其他特点

除上述这些之外，Slack 作为一款聊天系统还具备许多方便的功能和特征。

- 简洁的 UI

  Slack的UI设计简洁明了，让人看一眼就能上手使用。虽然目前（2014年11月）只提供了英文UI，但不懂英语的人用起来也不会有什么障碍。
- 可读性高的 text snippet

  用户可以用text snippet功能在聊天室内粘贴源码。这些源码会以语法高亮的形式显示。
- 可重复利用的消息记录

  Slack会给每一条消息设置永久链接，在引用消息内容时会经常用到这些链接。如果我们想引用某条消息，只要在Slack中粘贴它的永久链接，该消息就会显示在聊天框中。这一功能让我们的消息不再转瞬即逝，而是可以随时拿来重复利用。
- 类似书签的星标功能

  如果有想关注的聊天室或者怕忘记的消息，那么可以给它们标上星星。我们可以在星标一览中查看标记过星星的信息。有了这个功能，再久远的消息也能很快找到。
- 好用的搜索功能

  搜索聊天记录的功能拥有跨频道、跨群搜索以及指定日期等多种选择，能很轻松地找到

过去的信息或附件。

- 可添加自定义表情

  Slack允许用户添加自定义的表情用于聊天。我们可以添加上限为128像素×128像素的自定义表情，还可以添加公司或产品的Logo，这给我们带来了意想不到的便利。

◉ 收费版

前面我们说过，Slack 采用了免费增值模式，所以用户可以免费使用，但它同时还提供了收费版 Slack。收费版的特征如下。

- 付费

  收费版会根据活动用户数量收取费用。

- 可设置单频道来宾

  收费版可创建只能访问一个频道的免费来宾账户，也就是单频道来宾。在项目需要临时与团队外人员交流时，有单频道来宾会方便许多。

- 不限制记录保存数

  免费版只可以保存1万条聊天记录，但收费版没有这个限制。在交流频繁的时候，往往一天会出现几百条消息，1万条记录最多也就能撑半年。无法保存记录就意味着我们无法参考过去的消息，而收费版就不用担心这个问题了。

- 不限制 Integration

  免费版只能添加5个Integration，但收费版没有限制。

◉ Slack 应用

Slack 为 OS X、iOS、Android 准备了专用的客户端。只要在各个系统的应用商城中搜索 Slack 就能找到。

网页版的 Slack 不支持智能手机，所以 iOS 和 Android 要想使用 Slack 必须安装应用。iOS 和 Android 的 Slack 应用运行起来更快，用户能轻松查看消息通知和发送消息，而且新的消息通知会发送到 OS 的通知区域内，不用担心遗漏重要的事情。

截止到现在（2014 年 11 月），Slack 还没有 Windows 专用的客户端，但 Google Chrome 的“创建应用程序快捷方式”功能可以创建 Slack 的快捷方式，我们可以把它当成专用客户端来用。

4.3.3　Slack 做不到的事

但是，Slack 只能让我们与团队内部成员交流。当我们需要与团队外（公司外）的人进行交流时，必须将对方邀请进 Slack 才行。另外，有些时候让团队外的人使用 Slack 本身就是一件难事，因此可以考虑把 Slack 和普及率较广的 Skype 等软件结合起来使用，以便与团队外的人进

行交流。

Slack 还无法语音聊天。有些商讨内容打字聊天说不清楚，而语音会议的效果会好很多，这种时候就可以拿 Skype 等带有语音聊天功能的交流工具来配合使用。

4.3.4 Slack 的注册

Slack 的注册需要在 `https://slack.com/` 上进行。虽然注册全程为英语，但中间要输入的东西也就是邮箱地址、用户名、团队名、专用域名、邮箱域名、密码而已，并没有什么难的。创建团队后，Slack 会自动分配一个专用域名给我们（专用域名“.slack.com”）。此后，我们只要访问这个地址，就可以进行登录聊天室等操作了。

给团队设置邮箱域名后，所有拥有该域名邮箱的人都可以创建自己团队的账户。比如要在公司里用 Slack，只要我们在这里设置了公司的邮箱域名，公司成员就都可以创建 Slack 账户了。

4.4 对团队开发有帮助的工具

在前面几节中，我们了解了团队开发必不可少的问题跟踪系统、版本控制系统以及聊天系统。除此之外，还有很多能帮助开发团队共享信息和数据的工具。这里我们来了解一下其中的 Dropbox 和 Google Drive。

4.4.1 Dropbox

Dropbox[①] 是一项能轻松共享开发资料等文件的在线存储服务。用户不必准备文件服务器，只要通过 Dropbox 即可与团队成员共享文件。由于其前期工作只有注册用户这一步，所以可以省去搭建及使用文件服务器的成本。各位可以把它看作一个网络上的文件服务器。

使用 Dropbox 共享文件时不像邮箱或 Skype 那样烦琐，用户只需操作一下本地目录，文件就会自动完成同步，这可以为我们节省大量时间。另外，Dropbox 允许付费增加可用空间，而且为用户准备了个人与团体两个不同使用方案。

4.4.2 Google Drive

Google Drive[②] 是 Google 提供的一项云服务，用来创建和共享文档。它通过浏览器为用户提

① `http://www.dropboxchina.com/`

② `https://www.google.com/drive/`

供 Microsoft 的 Word、Excel、PowerPoint 等软件。其独特之处在于允许多人同时编辑一个文件，因此可以在文档中实现交流。此外，用户可以详细地为每个文件设置共享用户及权限。

4.5　小结

本章重点在于了解面向团队开发的工具，内容涉及问题跟踪系统、版本控制系统、聊天系统等。

如果我们能够灵活运用 Mercurial 和 Redmine，把握好开发成果及信息的共享环境，就可以有效提高开发效率。另外，借助 Slack、Google Drive 等工具，我们可以通过网络进行交流、共享信息和数据，开发工作不再受场所限制，减少了交通和交流方面的成本。

如果我们花些心思学会使用工具，还能有效地提高作业效率。动脑筋提高效率同样是一种乐趣。虽然团队开发中的交流更加复杂，但我们可以通过灵活运动工具提高沟通效率，从而做出更大的成果。

第 5 章　项目管理与审查

以团队形式进行开发时，应该遵循什么样的流程呢？首先，要分配作业任务，让每个成员负责一部分开发工作。每天或每周报告进度，如果工作过程中发现问题则必须马上共享。要想更好地管理这一系列流程，使开发工作更加顺利，我们要用到 Redmine、Trac 这类问题跟踪系统。

本章首先在 5.1 节就问题跟踪系统中的项目管理与问题的区分使用进行说明。另外，会在 5.2 节通过实际的示例讲解如何统一项目内容。

然后再进一步，将问题跟踪系统中的问题与版本管理系统中的分支相结合，为各位介绍问题驱动开发的相关知识（5.3 节）。

团队开发还有另外一项优势，那就是在分配任务和实现任务之余，还可以验证当前任务或实现是否正确地完成了，也就是所谓的审查。5.4 节将不惜大量篇幅，为各位讲解审查方和被审查方各自的注意事项，以及一些行之有效的审查方法。

5.1　项目管理与问题的区分使用

进行项目管理时，应该以什么为单位创建问题，每个问题中应该写什么，这都是整个团队必须统一的事。本节将以 Redmine 为例，学习进行团队项目管理时应注意的一些点。

5.1.1　项目管理的前置准备工作

在 Redmine 上创建项目、开始项目管理之前，有些设置需要先做好。我们可以在项目的“配置”标签页进行各类设置（图 5.1）。

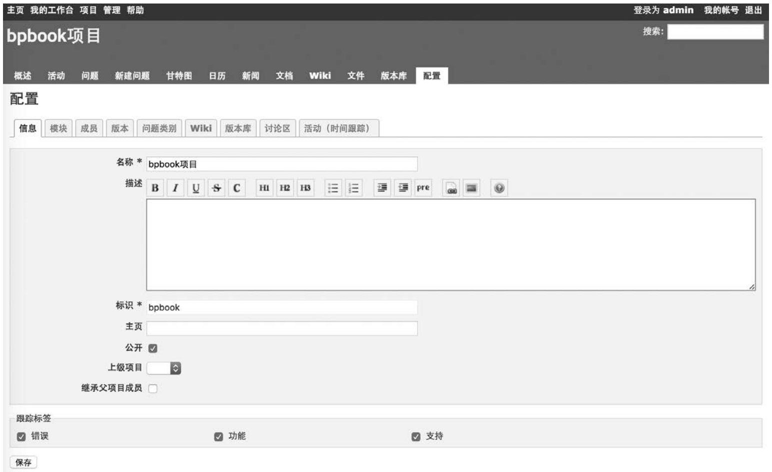


图 5.1　项目的配置界面

◉ 添加成员

首先添加参与项目的成员。成员按职责不同分为 3 种，具体区分如下。

- 管理人员：可更改项目的设置。一般情况下所有项目成员都可以设置为管理人员
- 开发人员：无法对配置界面进行操作，其余操作则不受限制。通常是公司外项目成员等
- 报告人员：测试负责人，用来对那些负责添加问题的成员进行设置

◉ 添加版本

一个项目必然存在确定需求、实现完毕、测试完毕等阶段点（里程碑）。我们要把各个里程碑的信息添加到版本标签页，将其视为一个个版本。如果版本发布日期已确定，则可以给相应版本设置日期。

◉ 添加问题类别

如果项目具备一定规模，那么最好以功能为单位添加问题类别。添加类别时，请记得设置该类别的负责人。在没有相应负责人的情况下，可以直接将项目组长设为负责人。这样一来可以避免出现没人负责的问题。

5.1.2　创建问题

现在我们已经做好了给项目创建问题的准备，可以实际动手创建问题了。点击“新建问题”标签会出现如图 5.2 所示的界面。

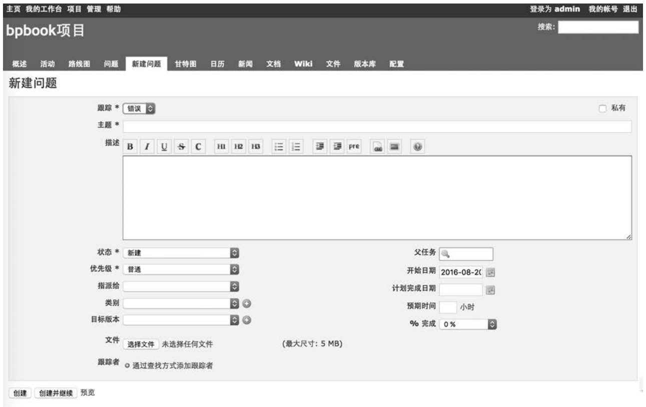


图 5.2 新建问题界面

创建问题时，各项目按如下内容输入、设置。

项目	内容
跟踪	指定问题的种类。区分如下所示[①]。 • 功能：用于实现新功能的问题 • 错误：用于报告 Bug • 任务：实际工作、操作等 • 需求讨论：用于讨论需求的问题。需求确定后新建功能问题并与之关联
主题	简洁明了地描述问题的内容
描述	描述这个问题想要实现的内容。用问题模板（Issue Template）统一描述内容后可有效防止遗漏
状态	设置问题的状态。比如开始工作后选择“进行中”，工作完成需要别人确认时选择“反馈”等
优先级	如果描述的内容比较重要，可以将优先级提升为“高”或其他
指派给	选择负责这个问题的人。如果不清楚该由谁负责，可以指定项目组长，再让项目组长更改为适当人选
类别	指定问题的类别，标明问题属于哪个部分、与什么功能相关
目标版本	如果项目中存在里程碑，则选择“目标版本”这一项
跟踪者	如果除负责人之外还有其他人员与此问题有关，则选择此项
文件	在此添加相关图片或文件
父任务	要与父任务相关联时，在此输入父任务的编号
开始日期	添加着手处理该问题的日期。可直接使用默认日期（创建日期）
计划完成日期	希望在哪一天之前完成此问题里的内容
预期时间	如果预估了工作时间，可写入此栏
% 完成	直接用 0% 即可

① 新建项目默认跟踪标签只有“功能”“错误”“支持”3 项，不过用户可以通过管理界面里的“跟踪标签”页面增加、删除和修改选项。这里的“任务”和“需求讨论”应该是作者自己创建的。——编者注

5.1.3　整理问题

在问题中描述工作内容，根据内容进行操作和实现，将完工的问题的状态改为“已关闭”。当这一系列周期顺风顺水时，我们会发现问题一个个减少，有一种项目向着成功迈进的感觉。

然而，因需求变更导致问题的内容与最新需求产生偏差，或是负责人设置不当等问题经常会导致开发无法进行下去，结果就是无人问津的问题越积越多。一旦问题堆积起来，我们将不知道自己还剩多少工作要做，自然也就看不到项目的终点。

为防止无人问津的问题越来越多，我们需要定期整理问题。问题的整理方法如下。

① 问题分类

按更新顺序重新排列未结束的问题。更新日期较早的问题大多已经停滞，所以可以根据日期从旧到新进行查看。

② 查看问题

查看问题里写的工作内容。确认工作内容与实施目标是否一致，以及负责人是否合适。

③ 结束、驳回问题

如果发现内容与其他问题重复，则注明对应问题编号并结束当前问题。如果工作内容与当前需求出现很大偏差，则写明理由并驳回该问题。

④ 修正问题

如果工作内容不明确或有错误，则要将内容修正至能正确进行为止。如果工作内容涉及面太广，同时出现多名负责人，则可以按照 5.1.4 节说明的方法进行分割。

⑤ 分配问题

将问题指派给合适的负责人。有些工作内容会涉及多名负责人，此时要根据工作优先级挑选出一名主要负责人。

要是等问题堆积起来再去整理，那我们很可能要在这上面耗上一整天，结果耽误本来该干的工作。建议各位平时就抽出一丁点时间来整理它们。

5.1.4　分割问题

当既有问题久久无法完成时，就需要我们重新审视该问题的目标了。因为我们在创建问题的时候，很可能没注意到其中暗藏的大量工作。一旦遇到这种情况，就应该把这些暗藏的任务分割到其他问题中。

这样一来，我们就可以把暂时不需要处理的内容向后放，或者将分割出来的问题交给其他负责人。另外，把每个项目都细分成问题还有助于管理。Redmine 能够给问题添加关联，比如设置父子关系，或者将几个问题设置为“相关问题”等，这对管理的帮助很大。

当然，我们也可以在最开始就把工作细分成较小的问题。这个分配工作既可以在着手该项目时进行，也可以等到必要时再做。重要的是，要根据工作量以及工作内容的复杂度合理分配任务，然后整理成问题。

5.2 问题模板

我们在 5.1 节了解到，新建问题时要在“描述”一栏描述这个问题想要实现的内容。不过真到新建问题时，我们往往不知道该写些什么。另外，一旦问题的描述不够充分，我们就要花费很多时间在留言交流上。

因此，为了防止问题的描述出现遗漏，团队应该事先统一好需要描述的项目。Redmine 有一个插件叫问题模板，它能生成统一的格式（模板）并把该格式反映到问题中。

这里我们将学习问题模板插件的使用方法，然后给各位看一个实际的例子。

5.2.1 安装插件

要想给 Redmine 的问题设置模板，需要安装 Issue Template Plugin[①]。

安装插件的流程如 LIST 5.1 所示。

LIST 5.1 安装 Issue Template Plugin

```
$ wget https://bitbucket.org/akiko_pusu/redmine_issue_templates/downloads/redm
ine_issue_templates-0.0.9.zip
$ sudo mkdir /usr/share/redmine/plugins
$ unzip redmine_issue_templates-0.0.9.zip -d /usr/share/redmine/plugins/
$ cd /usr/share/redmine
$ sudo chown -R www-data:www-data plugins
$ sudo cp -pr plugins/redmine_issue_templates/assets public/plugin_assets/redm
ine_issue_templates
$ sudo rake redmine:plugins:migrate RAILS_ENV=production
$ sudo bundle install
$ sudo service apache2 reload
```

如果插件安装正确，插件管理界面会显示 Redmine Issue Templates plugin，管理菜单中将多一条 Global Issue Templates（全局问题模板）（图 5.3）。

① http://www.r-labs.org/projects/issue-template/

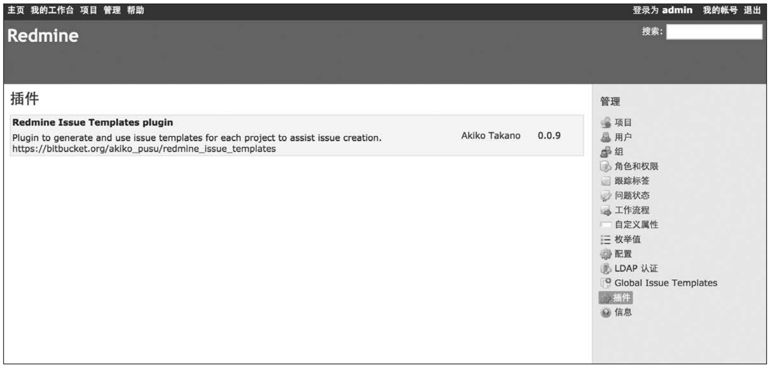


图 5.3　Redmine 的插件管理界面

5.2.2　问题模板的使用方法

下面我们来学习问题的设置和使用方法。

◉ 启用问题模板功能

打开项目的“配置”→“模块”界面，勾选“问题模板”，保存后即可对该项目开启问题模板功能。开启后，配置界面上将多出“问题模板”标签页。

另外，我们在这里更改全部设置，以便今后创建的所有项目都可以直接使用问题模板功能。打开管理界面的“配置”→“项目”界面，在“新建项目默认启用的模块”中勾选“问题模板”并保存（图 5.4）。



图 5.4　管理界面的项目配置

◉ 创建模板

在配置界面选中“问题模板”选项卡，点击“新建”进入模板添加界面（图 5.5）。

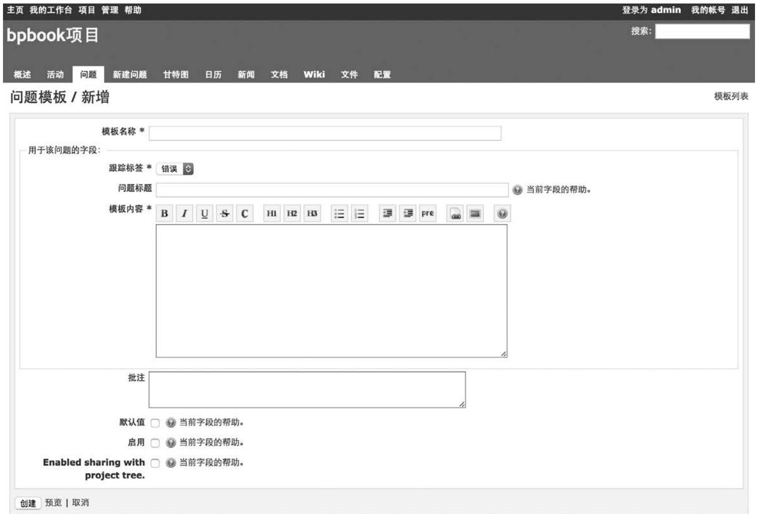


图 5.5　添加问题模板的界面

请按照下表设置各输入项目。

项目	说明
模板名称	在这里指定问题模板的名称。我们规范为“用于○○的模板”
跟踪标签	在这里指定应用模板的跟踪标签的种类（功能、错误、支持等）
问题标题	在这里指定标题的模板
模板内容	在这里输入问题内容模板。推荐使用标题或逐条列记，这种模板更直观易用
批注	不需要填写
默认值	勾选之后，选择该类问题的跟踪标签时会自动套用当前模板
启用	勾选之后，该模板才会生效
Enabled sharing with project tree.	勾选之后，子项目也可以使用该模板

◉ 套用模板

问题模板的使用方法很简单，只要在新建问题界面中选择模板就行了。

选择跟踪标签（问题类别）后，界面上会列表显示该跟踪标签下有效的问题模板。此处只要选择了任意一个模板，“主题”“描述”就会被自动填充。随后我们只需要按照通常创建问题的步骤，对内容作一些修改即可（图 5.6）。

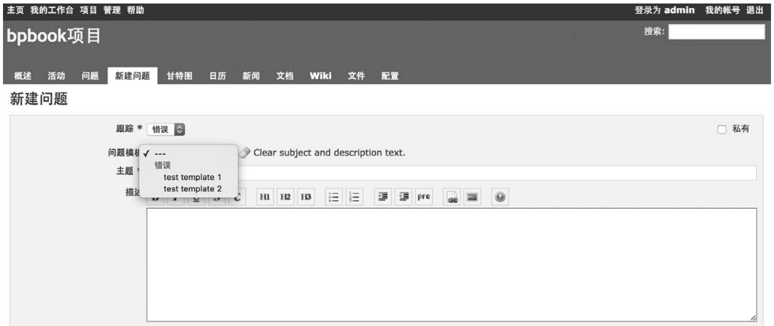


图 5.6　选择问题模板

5.2.3　Global Issue Templates

问题模板很方便，但当我们有一个所有项目通用的模板时，如果要一个项目一个项目地去设置这个模板，那可不是一般的麻烦。这种时候，我们可以使用 Global Issue Templates 功能，创建一个所有项目通用的模板。

创建 Global Issue Templates 时，先选择 Redmine 管理界面中的 Global Issue Templates 菜单，然后点击“新建模板”打开如图 5.7 所示的界面。这里的基本输入项目与问题模板相同，不同点则是下面多出了“项目”区域。这里会显示 Redmine 上的所有项目，我们在哪个项目前面打上勾并保存，该模板就会对哪个项目生效。

另外，如果我们新建了项目，Global Issue Templates 并不会自动对该项目生效。所以当我们创建了新项目时，需要打开 Global Issue Templates 的编辑界面，勾选新项目并重新保存。



图 5.7　添加 Global Issue Templates 的界面

5.2.4　问题模板示例

这里，我们为每个跟踪标签各准备了一个问题模板的范例。

我们用的格式是 reStructuredText，如果各位使用的是 Redmine 默认的 Textile，请自行作相应替换。

◉ 功能

与添加功能相关的问题（LIST 5.2）。描述要实现何种功能。

LIST 5.2　功能问题的模板

```
目的
====
- 实现该功能后成品能完成什么工作，将大致预想写在这里

输入输出
======
- 记录输入的值和输出的结果

相关功能、影响范围
==================
- 需要进行回归测试的地方
- 生成该功能所需数据的功能（可添加与对象问题的关联）
- 使用该功能所生成数据的功能（可添加与对象问题的关联）

安全
============
- 如有安全相关问题（权限等），则将相应内容写在这里

演示方法
========
- 记录 URL 或其他能轻松查看该功能的操作流程
```

◉ 错误

用于报告 Bug 的问题（LIST 5.3）。将试验等过程中发现的 Bug 制作成问题，分配给相应负责人。

LIST 5.3　错误问题的模板

```
现象
====
- 描述该 Bug 的现象
```

```
流程
====
- 描述出现该 Bug 的操作流程

预期结果
============
- 描述原本应该出现的结果

环境
====
- 描述试验环境

示例
 - OS：Mac OS X 10.10
 - 浏览器：IE11
 - 执行用户：admin
 - 出现 Bug 的 URL：https://staging.example.com/very-critical-feature

相关信息
========
- 描述错误日志、消息 Sentry 的 URL 等
```

◉ 任务

关于各种作业的问题（LIST 5.4）。

LIST 5.4　任务问题的模板

```
目的
====
- 执行该任务的目的

内容
====
- 具体要实施的内容
```

◉ 需求讨论

用来讨论需求的问题。确定需求后关闭这个问题，然后新建功能问题，并与之相关联。

LIST 5.5　需求讨论问题的模板（LIST 5.5）

```
目的
====
```

```
- 当前讨论的需求要达成何种预期。

需求方案
======
- 记录已有的需求方案，有多种时可记录多个方案
- 同时记录优点和缺点

总结
======
- 推荐的方案以及推荐理由

演示方法
========
- 记录 URL 或其他能轻松查看该方案的操作流程
```

5.3　问题驱动开发

接下来我们来学习问题驱动的团队开发。

5.3.1　别急着敲代码，先建问题

下面我们开始学习团队开发的一系列流程。所谓开发，既可以是开发新应用，也可以是修改已有系统。重要的是，我们在打开编辑器敲代码之前，要先把即将着手处理的项目添加到问题跟踪系统之中。

在 Redmine 界面中填入标题和任务概述并完成添加后，我们将得到一个带有编号的问题（这里是“#2”）（图 5.8）。

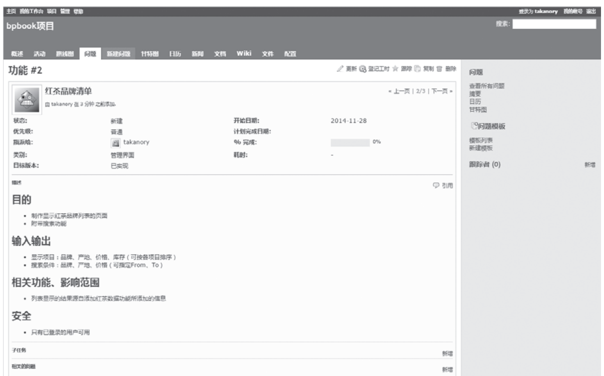


图 5.8　Redmine 的问题界面

5.3.2　创建与问题编号同名的分支

获得问题跟踪系统分配的问题编号之后，我们就可以在 Mercurial 上创建一个与之同名的分支了。举个例子，如果问题的编号为“#100”，那么我们的分支名就对应为“t100”。

这样一来，我们负责的项目和源码的修改内容就一一对应起来了。等到我们完成了这部分工作，只要将 t100 分支合并到 default 分支，就能轻松完成发布工作，十分方便。另外，将“一个问题对应创建一个分支”用作项目方针来规范团队，可以有效减少恼人的多头现象（Multiple Head，即最新端变更集分成多个版本的现象）。而且，由于问题的编号与分支名一一对应，所以当我们想查看项目 #100 所修改的内容时，只需查看分支 t100 即可，非常方便。

◉ 主题分支与问题驱动开发

这种为单一问题（主题）而创建的分支称为主题分支。

一个项目拥有的主题分支数往往很惊人。我们没必要给每个主题分支都起一个有意义的名字。花大量时间在起名字上，有时反而会起出一些让人摸不清头脑的名字，倒不如机械式地起名来得方便，而且不会出现歧义。问题跟踪系统的问题 ID 一般都很独特，再加上它具备我们前面提过的那些优势，所以用它肯定万无一失。

像这样基于问题跟踪系统的问题来进行开发的模式就称为问题驱动开发。

专栏　明确区分分支名与版本修订号

主题分支的英文是 Topic Branch，所以名字前面都加了 t。在 Mercurial 中，这还起到了区分分支名与版本修订号的重要作用。仅由数值组成的分支名很容易与版本修订号搞混，弄不好就会出现意外情况，所以应当尽量避免使用仅由数值组成的分支名。

5.3.3　让发布与分支相对应

从项目管理的观点讲，我们提倡设置里程碑并制定发布计划。而采用敏捷开发的项目还会基于迭代来制定发布计划。

在系统开发中，发布计划的重要性不言而喻。前面我们讲了问题与分支一一对应的好处，同样道理，将发布与版本管理系统的分支一一对应也能方便管理。

使用 Mercurial 时，default 分支是一个既定的分支，必然存在。因此我们规定“default = 面向正式运行的分支”。然后假设现在有一个发布，我们从 default 创建一个新分支与它相对应（LIST 5.6）。

LIST 5.6　从 default 创建发布分支

```
$ hg update default
$ hg branch m1231
```

然后，基于新创建的发布分支派生出各个主题分支（LIST 5.7）。

LIST 5.7　创建主题分支

```
$ hg branch
m1231
$ hg branch t100
```

用作里程碑的分支并没有严格的命名法则，但为了表示其“里程碑”之意，我们用字母 m 打头。

m 后面可以是发布日期（上例中的 1231 表示 12 月 31 日），也可以是明确表示发布的特殊词汇（比如智能手机版的发布命名为 m_smartphone 等）。由于问题跟踪系统不会专门给里程碑分配特殊 ID，而且创建里程碑并不像创建主题分支那样频繁，所以我们完全可以给每个里程碑设置个性化的分支名。

5.3.4　分支的合并

刚刚我们通过创建分支将开发成果分割开了。各个项目都有对应的分支，分支之间的开发和提交相互独立。因此，当项目完成后，还需要将它合并回来。合并的规则很简单，保证只合并有父子关系的分支就行了（图 5.9、图 5.10）。

① 只在父分支和子分支之间进行合并。
② 子分支之间不合并。
③ 从子分支派生出来的孙分支不能直接合并到父分支。

上述规则必须严格遵守。

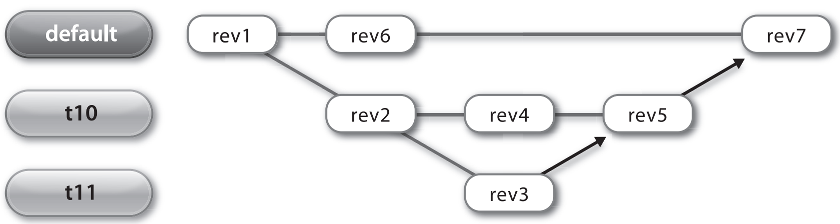


图 5.9　父分支与子分支之间的合并

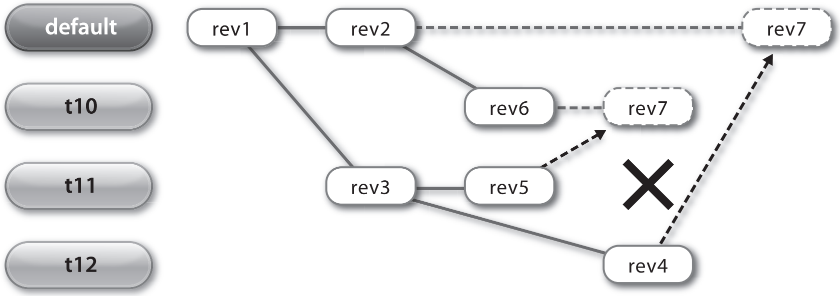


图 5.10　一些应当避免的合并

如果合并了没有父子关系的分支，我们好不容易分割清楚的项目会再度乱成一团。这可能会导致问题里没提到的修改内容在不知不觉间混入，或者导致源码产生冲突，等等。想在其他分支中反映某分支的成果时也要遵守这一规则，先与它们共同的父分支合并，再反映到对象子分支里。

虽然这样做很麻烦，但在父分支中分享一些本不该出现的半成品会导致很多问题，所以请各位务必遵守。

只要按照我们这里学习的方法创建分支，一般不会遇到子分支之间合并或者孙分支向父分支合并的情况。如果无论如何都得进行这类操作，那就可以认定是分割项目与设置父子关系的阶段出现了问题，需要我们修改问题的内容或选择其他方法应对。

另外，在将子分支合并到父分支之前，一定要先在子分支中反映父分支的修改内容。因为在我们修改子分支的过程中很可能有其他人对父分支作了修改，所以要先将这部分修改吸收到我们的子分支中，保证“父分支和子分支之间的差别 = 我们对子分支作的修改”。

这样一来，即便子分支与父分支之间出现了矛盾，我们只要回到自己的任务分支中就能解决该问题。

5.4　审查

5.4.1　为什么需要审查

审查的意义大致有两个。

一个是修正错误。通过审查人员的指正，我们能及早发现单人作业时忽略的问题以及一些错误的认识。

另一个是知识共享。审查能帮助团队成员分享在工作过程中学到的新知识，同时也是不同

特长的成员相互交流窍门和经验的好机会。

这里我们了解一下“受审查者在接受审查时”和“审查者在进行审查时”的注意事项。另外，本节将审查分为分“源码的审查”和“发布作业的审查”，分别称为代码审查和作业审查。

5.4.2　审查委托：代码审查篇

◉ 准备资料

接受审查前首先要准备资料。将自己写代码时参考的规格文档、需求定义文档、界面图等资料提供给审查者。

同时要让审查者清楚自己修改了哪部分源码。我们可以列出源码修改前后的差别，让审查者对修改内容一目了然。

◉ 统一编码风格

我们在第 1 章中了解到，PEP8 这种编码风格如今被人们广泛采用。所以我们可以先借助 flake8 统一编码风格，然后再请人来审查。flake8 的使用方法以及在编辑器中的设置方法请参考 1.3 节。

如果某个项目想采用与 PEP8 稍有不同的编码风格，我们可以准备出一个设置文件（setup.cfg）来自定义 flake8 的检测内容。

比方说我们想将一行的最大限制从 80 个字符提高到 120 个字符，那么要在项目根目录下的 setup.cfg 中作如下描述（LIST 5.8）。

LIST 5.8　setup.cfg 示例

```
[flake8]
 max-line-length = 120
```

> flake8 Configuration
> https://flake8.readthedocs.org/en/2.2.3/config.html

这样一来，flake8 就不会对 120 个字符以内的行发出警告了。变更项目的编码风格时，记得要先取得项目成员们的一致同意。

◉ 把希望审查员确认的事项整理出来

接下来思考一下审查时的步骤。漫无目的地检查全部代码既耗时又没效果，所以要圈出重点。

那么应该如何圈重点呢？请参考以下 2 个原则。

① 自己不放心的地方

处理比较复杂或者涉及陌生模块的地方等。总之要将自己不放心的地方交给审查者确认。

② 费心设计的地方

回顾一下实现过程中思虑再三的地方以及设计上比较费心的地方，将它们总结并整理出来。这些需要花时间思考的部分很可能是需求或设计上的重要部分，应该积极地交给审查者检查。另外，将编码或测试中的难点与其他团队成员共享也是一件有利无弊的事。

◉ 反映审查结果

接受审查时要虚心听取指正并记录下来。另外，根据指正进行修改后还要请审查者再确认一遍。

5.4.3　审查委托：作业审查篇

◉ 准备资料

接受作业审查时也需要准备相关资料。尽量把需要负责人严格把关的事故易发点共享出来，比如服务器的环境设置、网络构架等。

◉ 制作作业流程文档

将作业总结成“作业流程文档”。制作该文档时需要注意以下几点。

① 目的和概述

写出该作业的目的或概述。比如像下面这样的原因或问题经过、非正常作业流程等就要写出来，让审查者一目了然。

- 由于编号为 #123 的问题中的问题导致接单表数据不一致，为修正此问题进行数据维护
- 发布面向营业的接单数据报告功能。为进行发布作业而更改数据库定义以及修改 Apache 的设置

② 作业场所

写明是在哪里进行的作业。在正式环境由多台服务器共同组成时（比如主从式数据库）要格外注意这一点。

③ 所需时间与时间规划

各项作业的预计所需时间以及作业整体的时间规划能帮助我们制定当天的计划。另外，当外部相关人员加入时，我们还可以根据它来修改日程。

④ 负责人

如果事先知道当天的作业负责人，最好把负责人信息记下来。同时要写明进行该作业的

是团队内部人员还是承包方负责人。

⑤ 执行用户与权限

写明该作业由 OS 的哪个用户执行。这样就可以避免到了实际作业时才发现没有 root 权限而慌了手脚。这对写流程文档的人来说或许意义不大，但当天保不准会由其他人来代替我们进行该项作业，如果对方搞不清执行用户，将很容易引起麻烦。所以这一条还是记上为妙。

⑥ 执行命令

在命令行执行某条命令时，需要将该命令记录下来。只要保证作业内容固定，那么任何人执行该作业都能得到相同结果，造成问题的风险也就相对较小。另外，执行命令之后要检查命令是否正常执行完毕，这个检查的方法也最好写在这里。

⑦ 作业失败时的对策以及补救方法

对于一些有着复杂流程的作业，或者一旦失败就难以还原的作业，应该事先写明失败时的还原方法。事先准备好应对方法能避免正式作业中慌手慌脚。

⑧ 试运行的记录

如果在本地的 VM 上进行了试运行，那么应该把结果记录添加在这里。试运行的结果不但能让审查者对正式作业有一个更清晰的印象，还能帮助审查者发觉正式作业时需要注意的点。

审查者要根据这些资料判断受审查者的开发成果是否妥当。

5.4.4　实施审查：代码审查篇

◉ 形式上的检查没有意义

如果把有限的时间花费在检查“一行有多少个字符”“类或函数的定义之间空了几行”等问题上，那完全是在浪费审查时间。这些事项用 PEP8 或 pyflakes 等 lint 工具就能轻松搞定。

审查时要把时间花在以下几个项目的检查上。

① 程序是否满足需求

② 是否有遗漏需求

③ 设计或实现的效率是否够高

◉ 发现疑问就询问理由或缘由

在实际写代码的过程中，设计与实现上的很多判断都要基于各种背景或缘由。

受审查者在接受审查时不一定能把这些情况全都说清楚，而审查者又不知道对方漏说了哪

些知识，所以审查者在看代码时常常会觉得有些不对劲。作为一名审查者，我们必须考虑到这种情况。对于有疑问的部分，要向受审查者了解背景或缘由，然后努力去理解。如果我们有更好的解决方法，要当场提出。

◉ 整理注释

关于源码上是否应该留有注释，众说纷纭。不过，我们建议将修改程序时的背景或缘由简单地注释下来。日后再有人接触这部分源码时，很可能会对该项处理抱有疑问，而注释能帮助他们理解实现者的意图。

但这并不是说所有地方都要留注释。只要把实现过程中需要注意的难点顾及到了就行。另外，如果注释文的篇幅实在太长，最好改用文档或其他形式保存。

反过来，我们还需要检查该删除的注释是否都删除了。因需求变更而修改源码后，有一部分注释会与事实不符，这些注释就需要删去。

◉ 判断是否需要修正时，要考虑重要性和日程之间的平衡

即便我们断定受审查者写的代码需要修改，也要考虑修改内容的重要性以及日程。很多时候，代码修改后确实能提高可读性，但当涉及的代码规模很小时，我们完全可以不对本次代码作修改，只敦促对方下次注意即可。相对地，如果可读性差或效率低的代码已经被大量复制和粘贴，那就很可能对今后的修改造成影响，这时就必须督促对方修改。

不过，是否立刻修改仍有商量余地。有时考虑到日程所限，可以将修改程序的作业创建成问题并延后处理。

◉ 将编码能力不足和问题意识分开评价

有些时候，虽然受审查者的编码能力还欠火候，但需求和设计方面的问题意识可圈可点，一看就是认真考虑过的。受审查者自然希望获得肯定，所以这种时候我们要充分肯定对方的问题意识，只针对编码方面作出指正。

专栏　rietveld 的功能介绍

平日里，我们用 Mercurial 的 diff 进行简单审查，同时会还配合使用 Google 开发的代码审查工具 rietveld。这个专栏将对 rietveld 的功能作简单介绍。rietveld 是在 Google App Engine 上运行的 Web 应用式代码审查工具。

委托别人进行审查时，要将审查对象 diff 提交给 rietveld。提交内容将汇总为 patch set 形式，生成审查专用页面。根据 diff 的内容，该工具会生成审查对象文件一览表以及各文件的审查页面（图 5.11）。



图 5.11　审查对象文件一览表

各文件的审查页面有两种阅览形式（unified 形式和 side-by-side 形式）（图 5.12、图 5.13）。不管是哪种形式，在任意一行上双击都可以打开留言框，提交审查留言（图 5.14）。

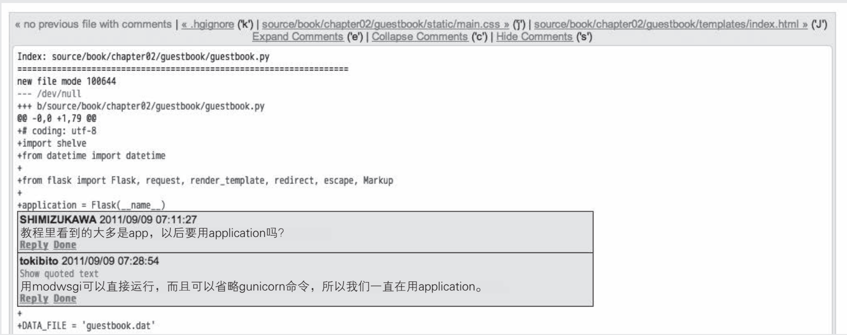


图 5.12　unified 形式的审查页面

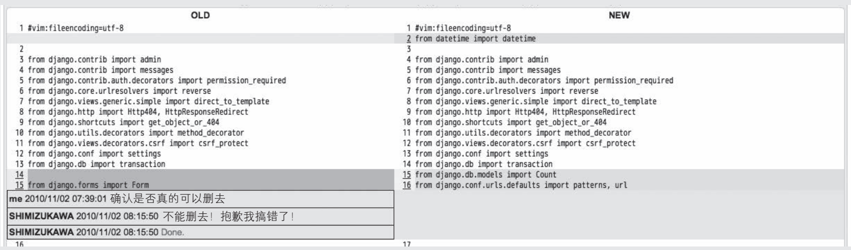


图 5.13　side-by-side 形式的审查页面

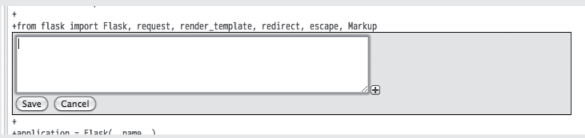


图 5.14　可以直接在代码中添加留言

> rietveld 这类工具虽然会增加审查的前置准备工作，但它可以高效审查涉及多个文件的大幅度更改，而且不必审查者和被审查者同时在场。根据审查对象的不同，我们可以选择用工具进行审查，或者用 diff 甚至是打印出来的源码进行审查，从而提高审查效率。

5.4.5　实施审查：作业审查篇

◉ 能否跟踪作业内容和实施历史

不论事前事后，我们都希望能够客观掌握负责人的作业情况，所以要督促对方将作业内容以流程文档等形式保留下来。在服务器或数据库上作业时则要记录已执行的命令或 SQL 语句。另外，命令的执行结果也要记录下来。这样一来，即便事后才发现作业失误或者作业内容不周全，我们也可以根据记录查找原因，制定对策进行弥补。

◉ 能否还原，是否有备份

作业失败时能进行还原是十分重要的。

比如这个作业要进行数据库的数据修正，如果我们发现事务忘记了 BEGIN，直接发出了 UPDATE 语句，那么必须指正出来。同样地，数据库备份的重要性也是不言而喻的。

◉ 是否受外部因素的影响

即便受审查者在写流程文档时制定了妥当的作业流程，实际作业仍可能受到一些与作业内容没有直接关系的外部因素影响而失败。

比如以下这些情况。

- 发布对象程序需要调用外部 API 时，API 方有访问限制，但程序没有申请解除限制
- 发布服务时，没有将维护消息通知给用户。或者维护时间过紧，每次都是勉强完成维护
- 忘记停止正式服务器的 cron 中设置的定时任务，在作业中误执行了批处理，从而引起问题

审查者要注意这类外部因素的有无。

5.5　小结

本章先讲解了使用问题跟踪系统时的注意点。

接下来对把项目管理与版本管理结合起来的问题驱动开发进行了介绍。问题与分支相互配合管理有助于开发顺利进行。另外，这样做还能有效减少源码的矛盾，让我们在团队开发时不

必为此费心。

至于审查，我们从被审查者和审查者两个角度出发，分别了解了其需要注意的地方。我们前面也说过，能够进行审查是团队开发的一大优势。但是，如果审查时抓不住重点，那就审查很可能沦为形式上的检查。希望各位能熟记本章讲述的这些注意点，去实践一些更有意义的审查。

第 6 章　用 Mercurial 管理源码

Mercurial 是我们公司标配的版本控制系统，公司的日常业务大多离不开它。

除前面几章提到的提交、push、pull、创建分支等基本操作之外，Mercurial 还具备许多功能。灵活运用这些功能能帮助我们在各种情况下顺利完成开发。

本章将重点讲解 Mercurial 的使用窍门，包括一些好用的功能、版本库的管理方法、各种工具等，帮助各位进一步熟练使用 Mercurial。

6.1　Mercurial 版本库的管理与设置

Mercurial 是分布式版本管理系统，所以原则上是不需要 Subversion 那种中央版本库的。但是，开发团队成员之间共享成果（变更集）时，如果仅凭对等（Peer To Peer）方式进行交流，效果是十分有限的。为解决这一问题，需要在所有开发成员都能随时访问的服务器上设置一个中央版本库，用它来共享成果。接下来将要介绍的，就是在服务器上设置中央版本库的步骤。

6.1.1　服务器上的 Uinx 用户群设置

要想对文件系统上的 Mercurial 版本库进行操作，必须能在对象版本库内的“.hg”目录下进行读写，这就需要相应的访问权限。因此，用版本库进行开发的 Unix 用户需要设置为同一用户群来集中管理。

我们创建名为 dev 的群作为本例的开发者群（LIST 6.1）。这需要以管理员权限执行 `groupadd` 命令。

LIST 6.1　创建 dev 群

```
$ sudo groupadd dev
```

然后用 `useradd` 命令给 dev 群新建用户，用户名指定方法如 LIST 6.2 所示。

LIST 6.2　新建 dev 群的用户

```
$ sudo useradd -g dev <username>
```

即通过 `<username>` 指定用户名。

NOTE

指定 -m 选项可以创建主目录。

指定 -s 选项可以设置登录时使用的 shell。

另外，向群中添加已有用户时要用 usermod 命令（LIST 6.3）。

LIST 6.3　向 dev 群添加已有用户

```
$ sudo usermod -a -G dev <username>
```

NOTE

用户群的修改内容不会对已登录服务器的用户立刻生效。直到下次登录时才会应用新的群。

6.1.2　创建版本库

由于服务器上要放好几个版本库，所以我们新建一个统一存放版本库的目录。本例的路径为 /var/hg/（LIST 6.4）。各版本库要在这个目录下创建。

LIST 6.4　创建 /var/hg/ 目录

```
$ sudo mkdir /var/hg/
```

接下来我们在这个目录下新建一个版本库。执行 hg init 命令，新建名为 testrepo 的版本库（LIST 6.5）。

LIST 6.5　创建 testrepo 版本库

```
$ sudo hg init /var/hg/testrepo
```

然后把版本库内“.hg”目录的所有权交给 dev 群，让 dev 群的用户能够对该版本库进行写入操作（LIST 6.6）。

LIST 6.6　将版本库的群改为 dev

```
$ sudo chgrp -R dev /var/hg/testrepo/.hg
```

另外还需要设置 SGID（Set Group ID），保证向版本库写入时，文件的所有者是 dev 群（LIST 6.7）。

LIST 6.7　设置 SGID

```
$ sudo chmod g+sw -R /var/hg/testrepo/.hg
```

这样我们就有了一个可多用户共同使用的版本库。

6.1.3　hgrc 的设置

要想在服务器端执行钩子（Hook）脚本，必须使用可信群或可信用户（关于钩子脚本的相关知识将在 6.2 节学习）。我们在 hgrc 的 trusted 节的 groups 中添加 dev 群。如果该文件不存在，则直接新建。至于 hgrc 的路径，假如版本库路径为 /var/hg/testrepo/，则 hgrc 位于 /var/hg/testrepo/.hg/hgrc。clone 来的版本库会自动生成这个文件，而用 init 创建的版本库并不会自动生成，需要手动创建（LIST 6.8）。

LIST 6.8　在 hgrc 的 trusted 节的 groups 中添加 dev 群

```
[trusted]
groups = dev
```

6.1.4　使用设置好的版本库

如果想通过 ssh 来 clone 我们刚创建的这个版本库，需要以“ssh:// 主机名 / 目录”的形式指定 clone 位置的路径。举个例子，假设服务器可通过 example.com 域名进行访问，现在我们想 clone 路径 /var/hg/testrepo 下的版本库，则命令如 LIST 6.9 所示。

LIST 6.9　对 example.com 上的 /var/hg/testrepo 版本库执行 clone 操作

```
$ hg clone ssh://example.com//var/hg/testrepo
```

指定路径时请注意斜杠数。协议部分 `ssh://`、域名 `example.com` 与目录 `/var/hg/testrepo` 的结合部分都是双斜杠。

NOTE

通过 ssh 连接时如果需要密钥，就需要在 ssh 客户端的设置文件 $HOME/.ssh/config 中预先设置好对连接对象使用的密钥文件。

6.1.5　使用 hgweb 建立简易中央版本库

Mercurial 具有 hgweb 功能，我们只需一个 `hg serve` 命令就能公开版本库浏览器的 Web 站点。

```
$ hg serve
```

启用 hgweb 功能的版本库默认允许 clone/pull，在 hgrc 中添加设置之后还可以进行 push 操

作，所以对于一些仅需要在 LAN 内共享的版本库来说，启用 hgweb 功能后就可以当作中央版本库来用，能省去不少功夫。

我们在“.hg/hgrc”中添加以下设置。allow_push 是允许经由 hgweb 进行 push 操作的 user 列表，加上 * 表示不需认证即可 push。push_ssl 用来设置 push 时是否需要 SSL。

```
[web]
allow_push = *
push_ssl = False
```

这里不妨加上 -d 选项，让 hgweb 的 Web 服务器常驻。启动之后会显示 hgweb 的 URL。

```
$ hg serve -d
listening at http://127.0.0.1:8000/ (bound to *:8000)
```

如果直接执行 hg clone 加 URL，本地版本库名会变成 domain:port，所以我们在 clone 时添加版本库名作为第二传值参数。

```
$ hg clone http://127.0.0.1:8000/ monjudoh
```

如下所示，我们在 clone 来的版本库中进行修改，然后提交并 push。

```
$ cd monjudoh
$ touch monjudoh.txt
$ hg ci -A -m "add monjudoh.txt"
adding monjudoh.txt
$ hg push
pushing to http://127.0.0.1:8000/
searching for changes
remote: adding changesets
remote: adding manifests
remote: adding file changes
remote: added 1 changesets with 1 changes to 1 files
```

6.2　灵活使用“钩子”

Mercurial 有钩子功能。所谓钩子功能，就是指 Mercurial 在执行特定处理时还能够额外执行其他处理的功能，而且这个额外处理是任意的。比如借助这个功能，我们可以在 push 时发送邮件通知，或者在提交之前自动检测提交是否有疏漏等。

本节将通过几个实际的例子，教各位灵活运用 Mercurial 的钩子功能。

6.2.1　钩子功能的设置方法

钩子功能的设置可通过在 hgrc 的 [hooks] 节下指定 shell 命令或 Python 函数来进行。想让哪个版本库执行钩子功能就在哪个版本库的 hgrc 中进行指定。比如在 6.2.2 节，我们想利用钩子功能在提交时检查编码风格，那么就在本地版本库的 hgrc 中进行指定；又比如在 6.2.5 节，我们想通过钩子功能禁止中央版本库出现多头现象，那么就要在中央版本库的 hgrc 中进行指定。

```
[hooks]
commit = echo commit;
update = python:myhook.sample
```

如果在一个钩子事件中指定了多个钩子脚本，那么将执行最后一个。

```
[hooks]
update = python:myhook.sample1
update = python:myhook.sample2 # 这个钩子事件会被执行
```

指定为空时不会执行任何操作。我们可以利用空指定覆盖已有操作，从而避免该操作被执行。

```
[hooks]
update = python:myhook.sample1
update =  # update 钩子事件不会被执行
```

在钩子事件名后面加上后缀，就能让同一个钩子事件执行多个钩子脚本了。

```
[hooks]
update.sample1 = python:myhook.sample1
update.sample2 = python:myhook.sample2
```

6.2.2　尝试钩子脚本

如今网络上有许多已公开的钩子脚本，这里我们以“提交时检验 PEP8 编码风格”的脚本为例进行导入。该脚本包含在名为 hghooks[①] 的钩子脚本集之中。

这个 hghooks 可通过 pip 安装。

```
$ sudo pip install hghooks
```

安装完成后，只要在 hooks 中添加以下设置即可开始使用。

① https://pypi.python.org/pypi/hghooks/

```
[hooks]
pretxncommit.pep8 = python:hghooks.code.pep8hook
```

6.2.3 钩子事件

钩子事件种类繁多，下面我们对应命令的执行类型，来了解一下其中的一部分。给版本库设置钩子功能时，要看对应命令是否会给版本库带来变更，这两种情况下所用的钩子事件是不一样的。

不会给当前操作版本库带来变更的命令有 update、从当前版本库 push、被其他版本库 pull 等。

会给当前操作版本库带来变更的命令有 commit、tag、从当前版本库 pull、被其他版本库 push 等。

接下来，我们对应着命令来了解一些已定义的钩子事件。其中，changegroup 表示 changeset 群被拿到版本库时的事件。另外，对 changeset 群中各个 changeset 执行的是 incoming。

◉ update 时

当前操作版本库的变更集不会被变更

① preupdate

② update

◉ 提交时

当前操作版本库的变更集会被变更

① precommit

② pretxncommit

③ commit

◉ 从当前版本库 push/ 被其他版本库 pull 时

向其他版本库发送变更集群的操作。当前操作版本库的变更集不会被变更

① preoutgoing

② outgoing

◉ 向当前版本库 pull/ 被其他版本库 push 时

接受其他版本库的变更集群的操作。当前操作版本库的变更集会被变更

① prechangegroup

② pretxnchangegroup

③ changegroup

④ incoming

我们将对版本库 A 执行各操作后的动作汇总成了下表。

执行的命令	版本库 A 发生的事件	版本库 B 发生的事件
hg update	preupdate, update	无
hg commit	precommit, pretxncommit, commit	无
hg push B	preoutgoing, outgoing	prechangegroup, pretxnchangegroup, incoming
hg pull B	prechangegroup, pretxnchangegroup, incoming	preoutgoing, outgoing

可以看到，对于部分会给当前操作版本库变更集带来变更的命令，其执行时发生的事件名带有 pretxn 前缀。这是因为这些变更是事务性的，钩子事件必须发生在变更处理已结束且事务尚未确定的时间点。

6.2.4　钩子功能的执行时机

接下来详细了解一下钩子功能。

我们看到，版本库发生变更时，有些钩子事件带有 pre 前缀，有些却没有。有 pre 前缀的钩子事件位于执行命令指定的处理之前，没有前缀的则位于执行命令指定的处理之后。pre 可以通过 exit 1 在命令指定的处理被执行之前直接中止命令。

比如我们把 /usr/bin/false 加入 preupdate。

```
[hooks]
preupdate = echo preupdate;hg parents;/usr/bin/false;
update = echo update;hg parents;
```

如下所示，如果工作目录的 parent 处于 tip（最新提交）的前一个状态，那么 `hg update tip` 命令在执行时将被 preupdate 打断。结果就是 update 并没有执行，工作目录的 parent 仍保持原样。

```
$ hg log -l2
changeset:   12:45648f583d32
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 14:03:11 2011 +0900

changeset:   11:2b1a3a2e6e29
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 14:02:55 2011 +0900

$ hg parents
```

```
changeset:   11:2b1a3a2e6e29
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 14:02:55 2011 +0900

$ hg update tip
preupdate
changeset:   11:2b1a3a2e6e29
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 14:02:55 2011 +0900

abort: preupdate hook exited with status 1

$ hg parents
changeset:   11:2b1a3a2e6e29
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 14:02:55 2011 +0900
```

版本库发生变更时，钩子事件有 3 种：有 pre 前缀的、有 pretxn 前缀的、没有前缀的。pre 位于命令指定的处理被执行之前，pretxn 位于处理实际执行之后且事务完成之前，无后缀的位于事务完成之后。

比如，我们这里设置 precommit、pretxncommit、commit 这 3 个钩子事件来执行 `hg tip`。然后执行 hg `commit` 会发现，从 pretxncommit 起 tip 就被更改了。

```
[hooks]
precommit = echo precommit;hg tip;
pretxncommit = echo pretxncommit $HG_NODE;hg tip;
commit = echo commit $HG_NODE;hg tip;
```

```
$ hg ci -A -m "commit successful"
precommit
changeset:   28:8a68c4364175
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 15:26:47 2011 +0900

pretxncommit d3000a3cea812e687f2f6bac7aaa2716be003cab
changeset:   29:d3000a3cea81
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 15:26:58 2011 +0900
summary:     commit successful
```

```
commit d3000a3cea812e687f2f6bac7aaa2716be003cab
changeset:   29:d3000a3cea81
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 15:26:58 2011 +0900
summary:     commit successful
```

不过，由于 pretxn 位于事务完成之前，所以在 pretxn 中用 exit 1 可以实现回滚。现在我们把 /usr/bin/false 加入 pretxncommit 中试试看。

```
[hooks]
precommit = echo precommit;hg tip;
pretxncommit = echo pretxncommit $HG_NODE;hg tip;/usr/bin/false;
commit = echo commit $HG_NODE;hg tip;
```

```
$ hg parents
changeset:   29:d3000a3cea81
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 15:26:58 2011 +0900
summary:     commit successful

$ hg ci -m "commit failed"
precommit
changeset:   29:d3000a3cea81
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 15:26:58 2011 +0900
summary:     commit successful

pretxncommit c42df85f2e7632b6e7ec124011b11ad8a9a3d61f
changeset:   30:c42df85f2e76
tag:         tip
user:        monjudoh <monjudoh@gmail.com>
date:        Fri Dec 02 15:36:52 2011 +0900
summary:     commit failed

transaction abort!
rollback completed
abort: pretxncommit hook exited with status 1

$ hg parents
changeset:   29:d3000a3cea81
```

```
tag:          tip
user:         monjudoh <monjudoh@gmail.com>
date:         Fri Dec 02 15:26:58 2011 +0900
summary:      commit successful
```

可以看到，pretxncommit 时反映出的提交已经被撤销。灵活使用 pretxncommit、pretxnchangegroup 能帮助我们验证提交的结果或 push 的结果是否有问题，并且在有问题时实现回滚。

6.2.5 编写钩子脚本

◉ 用 shell 脚本实现钩子脚本

我们以禁止中央版本库出现多头现象的钩子脚本为例，来了解一下用 shell 脚本实现的钩子脚本。首先如下所示，将脚本放到任意位置（“.hg”之下就不错）。

```
#!/bin/bash
# force-one-head
# add the following to <repository>/.hg/hgrc :
# [hooks]
# pretxnchangegroup.forceonehead = /path/to/force-one-head

if [ $(hg heads --template "{branch}\n"|sort|uniq|wc -l) != $(hg heads --temp
late "{branch}\n"|sort|wc -l) ]; then
    echo "There are multiple heads."
    echo "Please 'hg pull' and get your repository up to date first."
    echo "Also, don't 'hg push --force' because that won't work either."
    exit 1
fi
```

接下来，在 hgrc 中将其指定为 pretxnchangegroup 的钩子脚本。具体方法如下。

```
[hooks]
pretxnchangegroup.forceonehead = .hg/force-one-head
```

这个脚本被指定给了 pretxnchangegroup，所以它的执行时机位于 push 处理执行之后且事务完成之前。因此我们可以在 push 完毕的状态下执行各种 Mercurial 命令。这里我们利用 heads 命令检查是否出现多头现象，如果出现则通过 exit 1 回滚。这就是带 pretxn 前缀的钩子事件的用法之一。

在 Mercurial 上禁止中央版本库出现多头现象

http://labs.timedia.co.jp/2011/09/reject-multiple-head.html

◉ 用 Python 脚本实现钩子脚本

这里我们讲一个用 Python 脚本实现钩子脚本的例子，即在每次提交时都对第 2 章中完成的留言板发送一次 diff（LIST 6.10）。

LIST 6.10　myhook.py

```
# coding: utf-8
import logging
import urllib

# 接收方 URL
POST_URL = 'http://127.0.0.1:8000/post'

# 发送信息的格式
MESSAGE_FORMAT = """%(description)s
%(diff)s"""

def http_post(url, data):
    """ 用 POST 方法向 url 发送 data
    """
    fp = urllib.urlopen(url, urllib.urlencode(data))
    return fp.read()

def postdiff(ui, repo, hooktype, node=None, source=None, **kwargs):
    """ 将差别用 HTTP 进行 POST 的钩子函数
    """
    # 获取提交的上下文对象
    context = repo['tip']
    # 从上下文中获取差别列表（由于是迭代器对象，所以要列表化）
    diff_list = list(context.diff())
    # 将差别结合成文本
    text_diff = ''.join(diff_list)
    # 获取用户
    user = context.user()
    # 获取概要
    description = context.description()
    # 生成要发送的信息
    message = MESSAGE_FORMAT % {'description': description, 'diff': text_diff}
    # 生成发送数据的字典
    data = {
        'name': user,
        'comment': message,
    }
```

```
    # 发送
    http_post(POST_URL, data)
```

将 myhook.py 放到 PYTHONPATH 的影响范围之下，然后在 hgrc 中作如下描述。

```
[hooks]
commit = python:myhook.postdiff
```

提交后，会显示如图 6.1 所示的结果。



图 6.1　向留言板发送的提交信息

用 Python 写出的钩子脚本能直接使用 context 对象。当输出结果需要用版本库内数据进行条件判断等加工时，shell 脚本不但要处理输出格式，还必须对字符串进行操作。相对地，Python 脚本只需要从管理各修订版元数据的 context 对象中获取适当信息即可。

另外从管理方面上讲，Python 脚本可以利用 Python 的包管理，用 pip 就可以安装，十分方便。

6.3　分支的操作

下面我们来看看 Mercurial 对分支的操作。所谓分支，是指版本库中独立存在的开发线。分布式版本控制系统的一大优势就在于各个本地环境中的版本库互相独立。相对于 Subversion 等集中式版本控制系统，分布式版本控制系统能更放心地处理分支。

LIST 6.11　hg branch

```
$ hg branch
default
```

如 LIST 6.11 所示，默认情况下，版本库中只存在 default 分支。现在我们来创建一个新分支（LIST 6.12）。default 分支是版本库中原本就存在的分支。

LIST 6.12　hg branch（创建分支）

```
$ hg branch test-branch
$ hg branch
test-branch
```

分支创建出来之后，我们可以看到当前分支变成了 test-branch。不过对 Mercurial 而言，创建分支的流程到这里还没有结束。只有我们向新分支做过提交之后，这个分支才会具有实体。所以我们先向分支中添加文件，然后再看看效果（LIST 6.13）。

NOTE

Mercurial 允许用户在创建分支后直接进行提交，不必添加或删除文件。

LIST 6.13　向 test-branch 添加文件以及提交

```
$ touch test2.txt
$ hg add test2.txt
$ hg commit
```

至此分支创建完毕，我们来查看所有分支（LIST 6.14）。

LIST 6.14　hg branches（查看所有分支）

```
$ hg branches

test-branch   1:bcbc567db3dd
default       0:74471564b074 (inactive)
```

不出意外应该能看到 test-branch 和 default 两个分支。接下来我们回到 default 分支。

LIST 6.15　hg update（分支间的切换）

```
$ hg update default
```

如 LIST 6.15 所示，用 hg update 能在分支间进行切换。现在我们已经回到了 default 分支。下面来看看分支间合并的相关内容。

6.4　关于合并

利用 Mercurial 等版本控制系统进行多项工作时，必然离不开成果的合并。Mercurial 的 `hg merge` 命令能自动通过三路合并为我们完成合并工作，但并不是说所有情况下都能正常合并。本节，我们将学习基本的合并流程以及发生冲突时的应对方法。熟练掌握合并的相关技巧，能

让各位对版本控制系统更加得心应手。

6.4.1　未发生冲突的合并

多名成员并行开发时会发生多头现象。下面例子中就存在 rev1 和 rev2 两个头。要解决这一问题就必须进行合并。

```
$ hg log -G --style compact
o  2[tip]:0   c0d244a079ce   2011-11-18 11:40 +0900   tokibito
|    tokibito
|
| @  1   ead8ad7a4d9f   2011-11-16 11:07 +0900   monjudoh
|/     monjudoh
|
o  0   fe75405e6383   2011-11-16 11:05 +0900   monjudoh
     first commit
```

假设现在我们位于 rev1，进行合并要先运行 `merge` 命令再进行提交。在同一分支内合并多头时，`merge` 命令不需要加任何传值参数。

```
$ hg merge
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m "merge"
```

这样一来多头就被合并成了一个头。

```
$ hg log -G --style compact
@    3[tip]:1,2   b5f62f57038b   2011-11-18 14:07 +0900   monjudoh
|\     merge
| |
| o  2:0   c0d244a079ce   2011-11-18 11:40 +0900   tokibito
| |    tokibito
| |
o |  1   ead8ad7a4d9f   2011-11-16 11:07 +0900   monjudoh
|/     monjudoh
|
o  0   fe75405e6383   2011-11-16 11:05 +0900   monjudoh
     first commit
```

工作中我们也经常会遇到合并两个分支的情况。下面的例子是包含 default 和 tokibito[①] 两个

① tokibito 为本章撰写者冈野真也先生的 Twitter 用户名。——编者注

分支的状态。

```
$ hg log -G
o  changeset:   2:a663e1bf8f71
|  branch:      tokibito
|  tag:         tip
|  parent:      0:fe75405e6383
|  user:        tokibito
|  date:        Fri Nov 18 11:40:13 2011 +0900
|  summary:     tokibito
|
| @  changeset:   1:ead8ad7a4d9f
|/   user:        monjudoh <monjudoh@gmail.com>
|    date:        Wed Nov 16 11:07:59 2011 +0900
|    summary:     monjudoh
|
o  changeset:   0:fe75405e6383
   user:        monjudoh <monjudoh@gmail.com>
   date:        Wed Nov 16 11:05:34 2011 +0900
   summary:     first commit
```

此时就不能用无传值参数的 `merge` 命令了。因为分支内不存在多头，无法自动选择合并对象。

```
$ hg merge
abort: branch 'default' has one head - please merge with an explicit rev
(run 'hg heads' to see all heads)
```

假设我们位于 default 分支的 rev1，现在只要将合并对象的分支名交给 `merge` 命令就能完成合并。

```
$ hg merge tokibito
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
$ hg ci -m "merge"
```

执行后的合并情况如下。我们是在 default 分支下指定 tokibito 分支执行了 `merge` 命令，所以被合并的那个修订版现在属于 default 分支。这里千万注意别搞错合并方向。

```
$ hg log -G
@    changeset:   3:7dffb01df37b
|\   tag:         tip
| |  parent:      1:ead8ad7a4d9f
```

```
| |  parent:      2:a663e1bf8f71
| |  user:        monjudoh <monjudoh@gmail.com>
| |  date:        Fri Nov 18 14:17:13 2011 +0900
| |  summary:     merge
| |
| o  changeset:   2:a663e1bf8f71
| |  branch:      tokibito
| |  parent:      0:fe75405e6383
| |  user:        tokibito
| |  date:        Fri Nov 18 11:40:13 2011 +0900
| |  summary:     tokibito
| |
o |  changeset:   1:ead8ad7a4d9f
|/   user:        monjudoh <monjudoh@gmail.com>
|    date:        Wed Nov 16 11:07:59 2011 +0900
|    summary:     monjudoh
|
o  changeset:   0:fe75405e6383
   user:        monjudoh <monjudoh@gmail.com>
   date:        Wed Nov 16 11:05:34 2011 +0900
   summary:     first commit
```

6.4.2 合并时发生冲突以及用文本编辑器解决冲突的方法

虽然大部分情况下合并都能自动完成，然而一旦发生冲突，就需要我们来手动解决问题了。假设有如下多头情况。

```
$ hg log -G --style compact
o  2[tip]:0   0dffbb8f7780   2011-11-18 15:09 +0900   monjudoh
|    other
|
| @  1   a239fe812ab0   2011-11-18 15:08 +0900   monjudoh
|/     this
|
o  0   0648c3b5afbd   2011-11-18 15:07 +0900   monjudoh
     base
```

rev0、1、2（base、this、other）中 conflict.txt 的文件内容分别如下所示。

- rev0（base）

```
A
A
```

```
A
A
A
```

- rev1（this）第 3 行变更为 B

```
A
A
B
A
A
```

- rev2（other）第 3 行变更为 C

```
A
A
C
A
A
```

然后我们在 rev1（this）执行 `merge` 命令。

LIST 6.16　执行合并查看冲突

```
$ hg merge
merging conflict.txt
merge: warning: conflicts during merge
merging conflict.txt failed!
0 files updated, 0 files merged, 0 files removed, 1 files unresolved
use 'hg resolve' to retry unresolved file merges or 'hg update -C .' to abandon
```

不出意外应该会看到如 LIST 6.16 所示的消息。这就是发生了冲突的状态，我们来解决它。

首先执行一个查看冲突状态的命令。

我们用 -l 选项执行 `hg resolve` 命令，发现 conflict.txt 仍处于未解决状态（LIST 6.17）。

LIST 6.17　hg resolve -l（查看冲突）

```
$ hg resolve -l
U conflict.txt
```

U 表示“未解决”（Unresolved）。打开文件可以查看冲突的位置，具体请参考“查看冲突位置”部分的内容。这里与其他版本控制系统一样，可以手动进行修正。

◉ 查看冲突位置

我们打开合并时发生冲突的 conflict.txt 文件，会发现内容变成了下面的样子。其实简单说

来，所谓发生冲突，就是两个头对同一个文件的同一行进行了变更，计算机无法从机器逻辑层面上决定采用哪一方。要解决这个问题，就必须由人类明确表示出采用哪个变更。

```
A
A
<<<<<<< local
B
=======
C
>>>>>>> other
A
A
```

于是我们用文本编辑器打开 conflict.txt，手动编辑出合并后的 conflict.txt 文件。这里我们采用 rev1（this）的变更。

```
A
A
B
A
A
```

Mercurial 当然不知道人类是否解决了冲突，所以现在执行 `hg resolve -l` 仍然会出现之前的结果。此时我们要用 `hg resolve -m` 将当前状态从“冲突未解决”设置为“冲突解决完毕”（LIST 6.18）。然后只要提交即可。

LIST 6.18　hg resolve -m（解决冲突）

```
$ hg resolve -m conflict.txt
$ hg resolve -l
R conflict.txt
$ hg ci -m "merge"
$ hg log -G --style compact
@    3[tip]:1,2   c72367726805   2011-11-18 16:08 +0900   monjudoh
|\     merge
| |
| o  2:0   0dffbb8f7780   2011-11-18 15:09 +0900   monjudoh
| |    other
| |
o |  1   a239fe812ab0   2011-11-18 15:08 +0900   monjudoh
|/     this
|
o  0   0648c3b5afbd   2011-11-18 15:07 +0900   monjudoh
     base
```

6.4.3　合并的类型与冲突

上面我们提到冲突是“计算机无法从机器逻辑层面上决定采用哪一方”的状态，这里将对此作进一步的说明。

这里依然和前面一样，假设出现了如下多头现象。

```
$ hg log -G --style compact
o  2[tip]:0   0dffbb8f7780   2011-11-18 15:09 +0900   monjudoh
|    other
|
| @  1   a239fe812ab0   2011-11-18 15:08 +0900   monjudoh
|/     this
|
o  0   0648c3b5afbd   2011-11-18 15:07 +0900   monjudoh
     base
```

rev0（base）的 conflict.txt 文件的内容如下。

```
A
A
A
A
A
```

我们将 rev1、2（this、other）对第 3 行的修改情况以及合并后的结果总结成了下表（合并类型）。

合并类型

类型	this	other	结果
①	A	A	A（无变更）
②	B	A	B（采用 this 的变更）
③	A	C	C（采用 other 的变更）
④	B	B	B（碰巧合并成功）
⑤	B	C	冲突

①～④都是合并成功。①的双方都没有作变更，所以合并结果中也没有出现变更。

②和③都只有一方作了变更，所以只采用变更的一方。

④比较特殊，虽然双方都对同一行作了变更，但变更内容是相同的，所以可以直接采用。具体④的情况下能不能成功合并还要看版本控制系统的种类，虽然 Mercurial 认为是成功，但有一部分版本控制系统会判定为冲突。

⑤是双方对同一行作了不同变更，于是出现了冲突。

6.4.4　用 GUI 的合并工具进行合并

除命令行之外，Mercurial 还支持用 GUI 的合并工具进行合并。

◉ KDiff3

这里以 KDiff3[①] 为例介绍 GUI 的合并工具。KDiff3 支持 OS X、Windows、Linux，可以进行三路合并，是 GPLv2 的 OSS。

◉ 安装 KDiff3

任何环境下都能轻松安装 KDiff3。

○ 在 OS X 上安装

下载 dmg 并解压，将解压出来的 kdiff3.app 放到应用程序文件夹中。OS X10.9 无法直接使用放在文件夹中的 kdiff3.app。初次双击启动 kdiff3.app 时，系统会弹出对话框通知无法启动（图 6.2）。



图 6.2　通知 kdiff3.app 无法启动的对话框

于是我们需要以下流程。

初次启动时不要双击，要点右键通过上下文菜单打开（图 6.3）。



图 6.3　通过上下文菜单打开

此时会弹出确认启动的对话框，点击“打开”便能启动 kdiff3.app（图 6.4）。只要第一次成

① http://kdiff3.sourceforge.net/

功启动，以后我们就可以通过双击打开它了。



图 6.4　确认启动的对话框

○ 在 Windows 上安装

下载安装包并运行。

○ 在 Linux（基于 Debian）上安装

```
$ sudo apt-get install kdiff3
```

○ 在 hgrc 中设置 merge-tools

要想关联 KDiff3，在合并发生冲突时通过它来解决问题，需要在 hgrc 或 Mercurial.ini（Windows 的情况下）中作以下设置。

```
[merge-patterns]
**.* = kdiff3
[merge-tools]
# Override stock tool location
# MacOSX
kdiff3.executable = /Applications/kdiff3.app/Contents/MacOS/kdiff3
# Windows
# kdiff3.regkey=Software\KDiff3
# kdiff3.regappend=\kdiff3.exe
# kdiff3.fixeol=True
# kdiff3.gui=True
# Linux
# kdiff3.executable = ~/bin/kdiff3

# Specify command line
kdiff3.args = $base $local $other -o $output
# Windows
# kdiff3.args=--auto --L1 base --L2 local --L3 other $base $local $other -o $output
# Give higher priority
kdiff3.priority = 1
```

在 merge-tools 节的 kdiff3 中设置 KDiff3 的命令以及命令行对象，然后在 merge-patterns 节中设置所有文件冲突都使用 kdiff3 解决。

○ 用 KDiff3 解决冲突

我们还以 6.4.2 节描述的状态为例，给版本库执行 `hg merge` 命令。随后 conflict.txt 发生冲突，KDiff3 自动启动（图 6.5）。

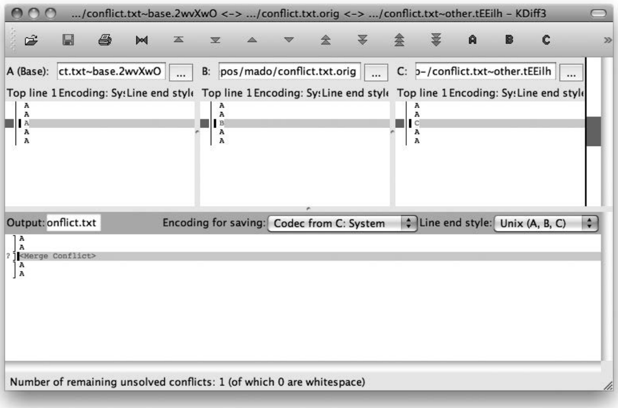


图 6.5 因发生冲突而启动 KDiff3 之后的状态

左、中、右的 A、B、C 中分别是多头共同的祖先版本、执行 `merge` 命令时的 parent 修订版、执行 `merge` 命令时的另一个最新修订版中的 conflict.txt 内容，下方的视图显示了合并完成后的结果。我们可以看到发生冲突的地方显示为红色。前面也说了，解决冲突就是要在发生冲突的位置明确选择采用（或者不采用）某一方的变更。

在 KDiff3 中，右键点击发生冲突的位置，就会显示菜单供我们选择变更（图 6.6）。

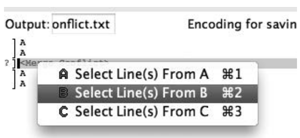


图 6.6 KDiff3 解决冲突时选择变更的菜单

这里我们选 B（图 6.7）。

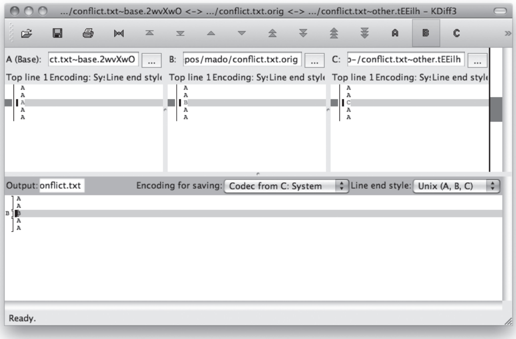


图 6.7　KDiff3 选择采用 B 时的状态

现在的状态是冲突已解决，界面中也明确显示出了合并对象的行采用了哪一个结果。随后保存文件并退出即可。

```
$ hg merge
merging conflict.txt
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

关闭 KDiff3 的同时 merge 命令也就执行完了。与未设置合并工具的合并不同，这里没有出现 Unresolved 的文件。最后只要提交一下，这次合并工作就完工了。

6.5　GUI 客户端

在前面的说明中，我们一直在使用 Mercurial 的命令工具（CUI 客户端），其实 Mercurial 也有 GUI 客户端工具的。本节我们将了解一下 GUI 客户端的优缺点，以及一些工具。

6.5.1　GUI 客户端的介绍

这里先介绍一些主要的 GUI 客户端及其导入方法。

◉ TortoiseHg

TortoiseHg[①] 最早是 Windows 专用的 GUI 客户端，如今已经可以在 Windows/OS X/Linux 上跨平台使用了（图 6.8）。

① http://tortoisehg.bitbucket.org/

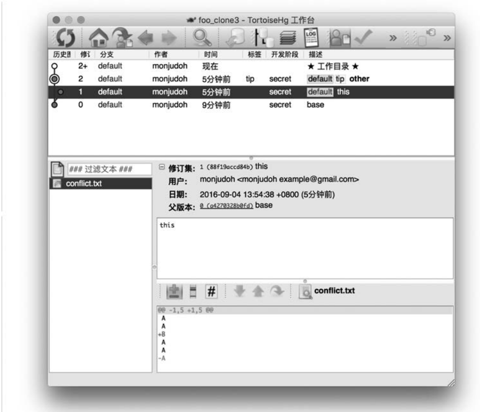


图 6.8 TortoiseHg 的示例界面

○ 安装

在 OS X 环境下，要通过 `http://tortoisehg.bitbucket.org/download/index.html` 的相应链接下载 zip 格式的 app 压缩包，解压后将 TortoiseHg.app 放到应用程序文件夹中并安装。之后的流程与安装 KDiff3 时一样。在 Windows 环境下，只要从 `http://tortoisehg.bitbucket.org/` 下载安装包直接运行安装即可。如果是 Linux，`http://tortoisehg.bitbucket.org/download/index.html` 也写了哪个发布版有程序包可用。如果各位的环境有相应程序包，可以通过 apt 等包管理系统进行安装。

◉ SourceTree

SourceTree[①] 是 Atlassian 提供的商用 GUI 客户端，同时支持 Mercurial 和 Git。最早是 OS X 专用的 GUI 客户端，现在则可以同时支持 OS X 和 Windows（图 6.9）。

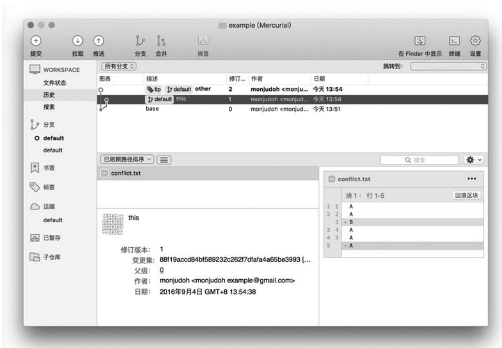


图 6.9 SourceTree 的示例界面

① `http://www.sourcetreeapp.com/`

○ 安装

如果是 OS X 环境，需要从 `http://www.sourcetreeapp.com/` 下载 dmg 文件，然后将解压出来的 SourceTree.app 放入应用程序文件夹并安装。Mac App Store 上只提供了旧版本，所以请不要用那个。如果是 Windows 环境，则要从 `http://www.sourcetreeapp.com/` 下载安装包直接安装。

至于合并工具的设置，在“环境设置”→“Diff”的“外部代码 差异对比 / 合并”部分进行（图 6.10）。



图 6.10　设置 SourceTree 的合并工具

6.5.2　GUI 客户端的优点

下面我们以 TortoiseHg 为例来看看 GUI 客户端的优势。

◉ 显示历史图

GUI 客户端最大的特点就是时常显示历史图，并且能以它为起点进行多种操作。

如图 6.11 所示，打开工作台，上部窗格显示历史图，左下窗格显示历史图中已选定的修订版与其 parent 之间的 status，右下的窗格则显示提交日志以及 status 窗格中已选定文件的 diff。在历史图上右键点击各修订版可以直接执行 update 等操作，十分方便。

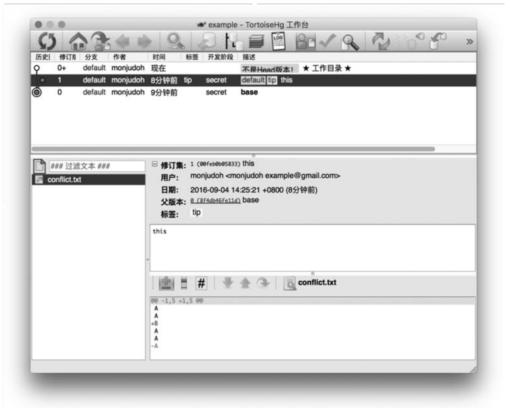


图 6.11　TortoiseHg 的工作台

◉ 工作目录的状态与差别的显示

在历史图中，工作目录与修订版享受同等待遇。选择工作目录后，左下窗格显示 status，右下窗格显示提交日志的草稿以及 diff（图 6.12）。

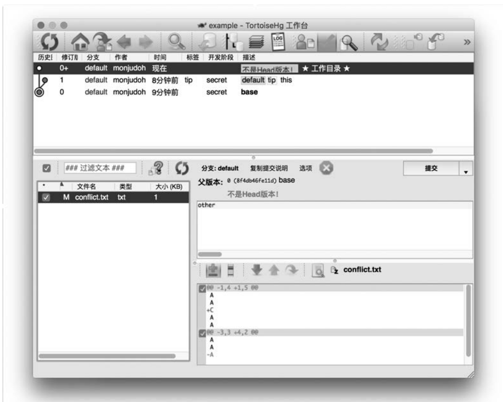


图 6.12　TortoiseHg 的工作台（选择了工作目录的状态）

提交也在这个界面中进行。由于提交时能看到文件复制的状态，所以能有效地减少提交方面的失误，比如出现了不该有的变更，或者在没有进行 add/remove 的情况下就进行提交等。

◉ 合并

○ TortoiseHg

TortoiseHg 会在合并发生冲突时显示 Unresolved 文件的列表。



图 6.13　TortoiseHg 解决冲突的界面

如图 6.13 所示，我们可以选择 Unresolved 的文件并点击右键，并从下述解决方案中进行选择。

- 通过 Mercurial 消除冲突
- 通常我们在合并时都选择了“尽量自动消除合并冲突”，所以不选用这个
- 启动 GUI 客户端消除冲突
- 采用当前所处位置的最新版本，并将状态改为 Resolved
- 采用合并目标一方的最新版本，并将状态改为 Resolved
- 编辑 Unresolved 文件后将状态改为 Resolved

6.5.3　GUI 客户端的缺点

GUI 客户端的缺点如下。

- 操作对象必须是 PC 本地版本库。通过 ssh 在服务器上工作时无法使用它们
- hg 命令只要导入 extension 就能轻松地添加功能，但 GUI 客户端只能使用其固定支持的 extension

虽说有着这些缺点，但几乎所有 GUI 客户端都提供了打开 terminal 的功能，在用户觉得用 CUI 更好的时候能随时切换到 CUI。至于在服务器上工作的问题，我们平时的提交可以通过 CUI 进行，遇到合并等用 GUI 更方便的操作时，可以先把版本库 pull 到本地再用 GUI 客户端处理。

6.6 考虑实际运用的 BePROUD Mercurial Workflow

多人合作进行开发时，难免会遇到几个本地环境之间的提交出现矛盾，以及为解决矛盾需要进行合并的情况。另外，在多种功能同时进行开发的过程中，很可能后实现的功能需要先发布。为防止这类情况给团队开发带来混乱，保证开发和发布的顺利进行，我们必须事先确定好版本库创建分支与合并分支的时机。

另外，我们的理想情况是所有工作人员都可以通过命令行对版本库进行操作，但有些时候受条件所限并不能满足这一要求。为在这种情况下也能保证工作进度，我们需要用到一些工具，这些工具也会在本部分进行介绍。

6.6.1 概述

这里将介绍的是我们自己使用 Mercurial 进行源码管理的工作流程和管理结构。我们将在讲述项目相关人员以及开发方法等背景的基础上，为各位说明工作流程的相关内容。

NOTE

本节内容以 bpmercurial-workflow 文档为基础。文档与源码的许可证为 CC[①] BY 2.1。本节讲述的流程与 Web 版相同，只是为了方便阅读对结构和文章作了少量修改。

bpmercurial-workflow 文档

`http://beproud.bitbucket.org/bpmercurial-workflow/ja/`

CC BY 2.1

`https://creativecommons.org/licenses/by/2.1/au/`

6.6.2 背景

既然要说明工作流程，那自然少不了项目的背景。所以我们先来了解一下背景。

◉ 目标项目

在这个工作流程中我们的目标项目是开发应用于 B to C 的 Web 站点（包括系统在内）。

多个小开发任务（包括设计变更等）并行进行，支持发布顺序不定的情况。

① Creative Commons，简称 CC，中国大陆正式名称为知识共享，是一个非营利组织，也是一种创作的授权方式。此组织的主要宗旨在于增加创意作品的流通可及性，作为其他人据以创作及共享的基础，并寻找适当的法律以确保上述理念。——编者注

源码等成品的交付工作，均通过 Mercurial 向客户管理的版本库（后述的发布版本库）进行 push。

◉ **这个工作流程的相关人员**

○ 程序员

提交系统的源码，主要负责管理版本库的人员。分支的合并与切换也由程序员负责。

服务器上已经为成员准备了各自的用户账户。为方便说明，我们这里假设程序员的用户名为 programmer。

○ 设计师

提交 HTML 模板、各种媒体文件的人员。也可能是网页制作工程师。

设计师需要向服务器上传文件，所以也备有用户账户。

○ 客户

接受开发完成的源码等成品的人员。通过后述的发布版本库取走源码。

◉ **工作环境**

程序员和设计师对源码进行修改、测试等工作的环境，即服务器或本地机器上的环境。

本书中，我们将在给开发专设的服务器（Linux）上对版本库进行操作。至于设计师工作的服务器环境，我们也已经保证该环境能启动应用程序服务器来查看模板文件等是否正常工作。以 Python/Django 为例来说，就是用 runserver 等来负责启动。

6.6.3　版本库的结构

我们使用的是分布式版本控制系统，所以必须掌握版本库的结构以及各版本库的作用（图 6.14）。

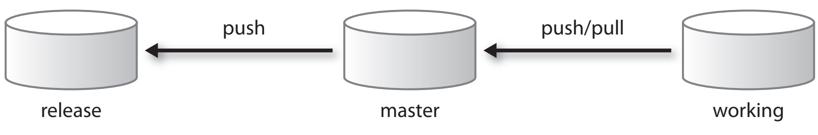


图 6.14　版本库的结构

◉ **主版本库（master）**

包含所有成果的版本库。用来积累以及共享整个团队每天开发出来的成果（比如开发中的分支等）。

版本库路径示例：/var/hg/example-prj

◉ 发布版本库（release）

装有已完成开发且等待发布到生产环境中的源码。放入这个版本库中的代码就是我们交给客户的成品。客户会使用这个版本库中的代码进行部署等工作。一般只有 default 分支处于活动状态。

进行发布（交付）工作时，我们要把 default 分支从主版本库 push 到发布版本库。

版本库路径示例：/var/hg/example-prj-release

◉ 工作版本库（working）

进行源码添加及修改等工作的版本库。开发成果要先提交到这个版本库，然后再 push 到主版本库。获取其他人的开发成果时，要从主版本库 pull 到这个版本库。

工作版本库既可以在服务器上也可以在本地环境上。

由于本书所介绍的开发将在服务器上进行，所以我们将主版本库 clone 至工作人员在服务器上的主目录下。

版本库路径示例：/home/programmer/example-prj。

6.6.4 提交源码

源码写好后要提交到工作版本库。

default 分支要时常用于发布，所以在提交成果时，除了为发布而进行合并以外都要使用其他分支。

◉ 问题与分支

我们在第 5 章中也提到了，修改源码时要先发起问题，然后根据问题编号创建相应分支。

○ 示例

为了修改源码，我们在问题跟踪系统中创建了编号为 #5 的问题（图 6.15）。



图 6.15 在问题跟踪系统中创建的问题

这种情况下，修改后的源码要提交到 t5 分支。于是我们新建 t5 分支。

```
$ cd ~
$ cd example-prj
$ hg branch  # 查看当前分支
default
$ # 问题编号为 5，所以从 default 分支创建名为 t5 的分支
$ hg branch t5
```

接下来，将源码添加成为管理对象，然后提交。

```
$ hg add
$ hg commit
```

这样就可以将我们的修改提交上去了。

在这个状态下，default 分支并没有被修改，所以就算其中的代码传到其他版本库（比如发布版本库等），我们所作的修改也不会被发布（图 6.16）。



图 6.16　向分支提交之后的历史图

一般说来，修改过的源码不会很快被发布。

如果我们将暂不发布的源码提交到了 default 分支，那么一旦遇到需要插入临时发布的情况，就必需面临很多容易出问题的操作，比如多头现象、删除不合适的变更等。

只有到了发布那一刻才能将源码提交至 default 分支，也就是在开发过程中保证源码只出现在开发分支里，这样才能保证安全。

我们将工作版本库的更改传到主版本库，具体代码如下。

```
$ hg push --new-branch  # 指定 new-branch 远程添加分支
```

专栏　hg push --new-branch

不加任何选项的 hg push 命令不能在远程版本库创建新的头。不但已有的分支受此限制，而且在远程版本库新建分支的操作也包含在内。指定 --new-branch 选项可以在远程版本库添加

新分支。另外，即便指定了 --new-branch 选项，我们依然无法给已有分支创建新的头，所以可以放心大胆地创建新分支并 push 变更。还有一点要注意，就算我们已经将本地创建的工作分支与 default 等合并，在 push 时仍然需要指定 --new-branch 选项。

6.6.5　提交设计

如何对待设计模板以及媒体文件是个非常难的点，而且根据设计师的技术以及工作人数的不同，这项工作的难度会有极大变化。一般说来，我们会使用 GUI 客户端通过 FTP、SCP 等将文件上传至服务器，确认其正常工作后再直接在服务器上提交文件。

◉ 在分支中进行设计

设计模板和媒体文件也要和源码一样提交到分支中。这是为了保证在与系统合并失败时能立刻还原到正常工作的状态。至于分支名，我们建议也和源码一样，起一个与问题编号相对应的名字。

如果设计需要频繁更改但系统没有变更，那么上面的方法会显得很繁琐，工作量很大。这种情况可以创建一个设计专用的分支，把所有和设计相关的变更都提交到这里（图 6.17）。

```
$ cd example-prj
$ hg branch
default
$ hg branch design  # 从 default 创建设计专用的分支
$ hg commit  # 提交
```



图 6.17　向设计专用分支提交后的历史图

◉ 伴随系统变更而产生的设计提交

在添加、修改系统源码之后，如果需要对设计进行添加或修正，那么设计师必须先将查看设计的分支切换至已提交系统源码变更的分支，然后再进行自己的提交。

6.6.6　分支的合并

◉ 发布时的合并

为发布（交付）在工作分支中开发的功能或设计，我们需要先将工作分支合并到 default 分支（图 6.18）。

```
$ hg update default
$ hg branch
default
$ # 为发布 t5 分支中的变更，要将该分支合并到 default 分支
$ hg merge t5
$ hg commit
```



图 6.18　t5 分支合并到 default 分支后的历史图

◉ 用来追踪最新变更的合并

发布之后，未发布的分支也必须吸收这些变更，也就是要将 default 分支合并到对象分支（图 6.19）。把 default 分支的成果吸收到所有对象分支中能带来很多好处，比如对象分支向 default 分支合并时不会出现冲突。另外，对象分支的头与合并后的修订版，即 default 分支的新头之间不会有任何差别。这意味着只要合并前的分支通过了测试，合并后也必然能通过测试。

```
$ hg update t3
$ # 将 default 分支合并到 t3 分支，使 t3 分支吸收最新的代码
$ hg merge default
$ hg commit
```



图 6.19　default 分支合并到 t3 分支后的历史图

专栏 如何处理已经无用的分支

当我们完成某部分开发，将与问题相对应的分支合并到 default 之后，这个分支就没有用了。但是，它仍然会以（inactive）的形式残留在 Mercurial 的分支一览中。

```
$ hg branches
default            7:a587bc0eb68e
design             6:e3f875bb6623
t3                 3:0440d428d18a
t2                 2:c4b2cd02f0c5
t1                 1:c65aded0fe16
t5                 4:9647e05b7580 (inactive)  # 已经合并到default的分支仍会留在一览表里
```

inactive（非活动）状态表示该分支已经合并到其他分支（这里是 default）了。因此，有些尚未被发布的分支也可能进入 inactive 状态。如果不分清已经发布完毕的无用分支和仍要使用的分支，就很可能酿成遗漏发布等事故。Mercurial 有一个专门表示无用分支的状态 closed（已关闭），我们可以将没有用的分支设置成这个状态。关闭某个分支（转入 closed 状态）时，需要先切换到该分支下，然后在提交时指定 --close-branch。

```
$ hg update t5
$ hg branch
t5
$ hg commit --close-branch  # 将t5分支转为closed状态
$ hg branches  # 已被close的t5不再出现在一览表中
default                        7:a587bc0eb68e
design                         6:e3f875bb6623
t3                             3:0440d428d18a
t2                             2:c4b2cd02f0c5
t1                             1:c65aded0fe16
```

closed 状态的分支虽然不会显示在 hg branches 命令的一览表中，但这并不代表分支被删除了。虽然不关闭分支不会对开发造成影响，但考虑到事故风险，还是建议各位将无用的分支关闭。

6.6.7 集成分支

在统合多个分支的变更内容时，我们要创建一个集成分支来合并它们。

◉ 创建问题

先创建问题来确定分支编号（图 6.20）。



图 6.20　集成分支的问题

◉ 创建集成分支

创建与问题编号相对应的集成分支 t6。

```
$ hg update default
$ hg branch t6
$ hg commit   # 提交分支
```

将要集成的对象分支合并到 t6（图 6.21）。

```
$ hg update t6  # 更新集成分支
$ hg merge t1  # 合并 t1 分支
$ hg commit
$ hg merge t3  # 合并 t3 分支
$ hg commit
```



图 6.21　将对象分支合并到用于查看的分支后的历史图

发布集成分支中的全部内容时，要将集成分支 t6 合并到 default 分支。

```
$ hg update default
$ hg merge t6  # 集成分支
$ hg commit
```

专栏　集成分支的用武之地

在变更内容出现冲突，或者需要在特定时间整合分支并统一发布等情况下，集成分支会显得非常好用。

另外，我们还可以将开发完成的各功能分支先合并到集成分支，经过综合测试之后再向 default 合并。

6.7　小结

本章就 Mercurial 的使用方法进行了说明，内容如下。

- 钩子功能的概述与活用方法
- 分支、合并以及消除冲突的方法
- GUI 客户端的介绍

另外，还对多人开发时的工作流程进行了说明，它对于防范冲突及运用上的不统一有着重要的意义。

如今，无论人多人少，Mercurial 等版本控制系统都是开发中必备的工具，能否用好版本控制系统已成为左右开发效率的一个重要因素。

专栏　Git 与 Mercurial 的区别

Git 是一种广受欢迎的分布式版本控制系统，与 Mercurial 齐名。因此，我们能看到很多 Git 与 Mercurial 的命令对照表，这就是为 Git 用户准备的 Mercurial 入门，或者是为 Mercurial 用户准备的 Git 入门。不过，如果让 Git 用户单纯根据命令对照表来使用 Mercurial，或者反过来让 Mercurial 用户根据对照表使用 Git，恐怕会是一种很危险的行为。要知道，虽然 Git 和 Mercurial 同为分布式版本控制系统，但二者的模型并不相同。乍看上去，人们只要记住 Git 中的命令和 Mercurial 中的命令的差别，就能随便在二者间换着用了，然而即使有对照表，也无法改变它们不同的本质，所以对很多操作都是都会产生误解，这早晚会让我们栽跟头。举个例子更能说明 Git 和 Mercurial 模型间的差异，Mercurial 的版本库由提交对象的图表构成，而 Git 的版本库由提交对象的图表和引用构成。

这个差别最明显地体现在对分支的看法以及同步（push/pull）上。

Mercurial 的分支本质上是给提交对象的分支的根部命了名。严格说来，分支只不过是被命了名的各个提交对象的参数。但是，在我们给子分支新命名之前，子分支会继承父分支的名字，

所以说它是在给分支的根部命名。

相对地，Git 的分支是给提交对象的图表的最新端命了名。这是一个每次提交都会跟着最新端切换的引用。Git 的 fast-foward 合并其实就是这个表示分支的引用的切换，分支的删除其实就是引用的删除。因此，Mercurial 中既没有 fast-foward 合并也没有分支的删除 (fast-foward 曾被导入过，但与 Git 的 fast-foward 合并仍是貌相似而质不同)。

如果把 push/pull 看作版本库的同步，那么版本库模型不同的理由将更加显而易见。Mercurial 的版本库是提交对象的图表，因此只需将提交对象图表的差别同步即可。push 操作中只需发送对方没有的那部分图表，pull 则只需获取自己没有的那部分图表。比如“删除历史”就无法同步。就算我们加上 -f 选项，最多也就是让历史受到污染 (共享了多余的子图表)。

Git 的版本库是提交对象的图表和引用，因此双方必须保持同步。切换引用的必要性致使 pull 需要伴随合并操作。如果不进行合并，该分支就将出现 remote 的和 local 的两个最新端。但是 Git 的分支是给最新端命的名，这就要求一个分支只能有一个最新端。Git 的 pull 之所以必须以分支为单位，恐怕就是因为存在这样一个需主观意识判断的合并过程。push 无法进行主观意识判断，所以只有满足 fast-foward 合并 (不需主观意识判断的合并) 的情况下才允许 push。

在更改历史上也有显著差异。Git 在日常使用中充满了更改历史，所以习惯 Git 的人在用 Mercurial 时总会去找类似的操作。但是，此时直接按对照表操作会出现很大隐患。虽然 Git 和 Mercurial 中都有“更改历史”，但二者实际要做的事却完全不同。Git 是新建图表并将引用切换到最新端，所以在 GC 启动之前仍然保留着更改前的图表。这个图表我们只是看不到罢了，但仍可以通过 reflog 引用它。相对地，Mercurial 是实际更改原有图表，即向图表添加新的子图表并删除旧的子图表。一个由图表的引用构成，一个由图表直接构成，这里，二者的差别非常显著。正因为如此，Git 在更改失败时只需引用 reflog 寻找更改前的修订版 (分支的最新端)，将这个修订版设置为该分支的头即可完成恢复，但 Mercurial 就必须通过 bundle backup 恢复旧的子图表，然后用 strip 命令删除新的子图表。可见，由于 Git 与 Mercurial 在模型方面存在差异，导致更改历史有着实质上的不同，更改历史失败后的恢复难度及安全性也都大相径庭。所以各位在用 Mercurial 更改历史时，建议先让保存原历史的选项有效，等确认更改无误后再通过 strip 命令删除旧的子图表。

第 7 章　完备文档的基础

这世上有着数不尽的项目，其中既有工作项目，也有爱好者们开发的开源项目。但是，这些项目中有相当大一部分都没有写文档，或者事后才补写文档。为什么开发者们不喜欢写文档呢？这其中包含着怎样的问题呢？

本章中，我们将考察何种环境能便于开发者写文档。同时作为例子，还将介绍文档编辑工具 Sphinx。

7.1　要记得给项目写文档

项目文档的内容分为许多种。虽然我们不必在一开始就把所有文档都写出来，然而一旦缺少关键文档，就很可能造成项目止步不前。

《敏捷建模：极限编程和统一过程的有效实践》[①]（Scott Ambler 著）一书是这样回答“应该何时写文档”这个问题的。

- 据我的经验，当实际需要时再去做模型或写文档比较有效
- 只在遇到麻烦时才更新文档
- 所有文档都应该尽量晚写，应该留到马上要用时再写

所以我们需要的是写文档的时间计划，以及想写随时就能写的环境。那么，什么样的环境能让人随时可以写文档，什么样的环境又能让人有动力写文档呢？或者反过来说，妨碍我们写文档的因素都有哪些呢？下面我们就从 Python 程序员的视角出发，考察妨碍以及促使我们写文档的因素。

7.1.1　写文档时不想做的事

回顾我们以往的经历，那些没有写文档的情况都源于某些共通的理由，下面就介绍其中几个理由。

◉ 不想用写文档的专用工具

写文档时常用的工具有下面这几种。

---

① 原书名为 *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*，Scott W. Ambler / Ron Jeffries 著，张嘉路译，机械工业出版社，2003 年 1 月出版。——译者注

- 文字处理软件
- 电子制表软件
- Wiki
- 其他综合型工具

这些工具在使用上有许多地方都和我们平时编程用的编辑器不同。为了写文档，必须用自己用不惯的编辑器或工具，这成了写文档时的一大制约。我们更希望能用平常编程用的编辑器写文档。

◉ 不想编辑流水账似的单一文件

一般的文字处理软件都是一个文件管理一篇文档。如果一个文件纵向延伸得较长，我们在编辑或显示内容时就需要将其滚动到特定位置。写文章过程中一旦需要对其他地方进行修改，就必须先滚动到待修改的位置再滚动回来。这种操作往往每写一篇文档就要经历许多次，而且每次寻找原先的位置都很麻烦。

然而这并不是说给工具加个界面分割或是记忆位置的功能就能行的。问题的根源在于一个文档以一个大文件的形式被管理着。这就像一个程序员肯定会觉得可分割的源码写在一个文件里十分影响效率一样。

要解决这个问题，应该像源码一样将文档分割成多个文件管理。

◉ 不能用 Mercurial 等管理差别，让人生厌

用文字处理软件写出的文档是单一的二进制文档，不管我们在里面修改了几个字、几段文章或者几幅图片，Mercurial 等版本控制系统都无法掌握该文件中的被修改之处。另外，如果有好几个人编辑同一个文件，很容易出现变更冲突，而且无法事后自动整合双方的变更。

由于存在着这些问题，我们很难放心大胆地去添加变更。

◉ WYSIWYG 工具在装饰和显示上很费时间

WYSIWYG 是 What You See is What You Get 的简写，意思是“所见即所得”。顾名思义，用 WYSIWYG 类工具写文档时能直接看到文档最终的显示或装饰效果。

不过，这有时也会让我们在写文章时分心去考虑显示效果，在装饰和显示上花费多余时间。

◉ 不想把参考资料和程序分开写

极限编程等敏捷过程问世以来，人们对程序的概念发生了变化，那就是从“程序是按设计书写的一成不变的东西”变成了“程序是阶段性变化的东西”。

在程序阶段性变化的过程中，文档必须跟着程序的脚步进行更新。如果文档没能跟上程序的变化，看文档的人就会按照陈旧的错误信息去写程序，导致开发无法顺利进行。

其实，人们很早以前就开始借助专门工具来解决这一问题了。这些工具可以自动将函数定义附近的 API 文档反映到参考文档中。但可惜的是，有些编程语言无法使用这类工具。

所以我们需要一个轻松的联动机制，保证 API 和函数的实现与参考文档的内容不出现错位。

◉ 不想写没人看的文档

没人看的文档根本没必要写。那么，我们应该如何分辨一个文档有没有人看呢？

写出来的文档没人看，主要原因是这个文档的目的不够明确，让人不知道是写给谁的，为什么写的。所以在项目的早期阶段就该给文档做好计划，规定好什么时候写、为了什么目的而写，以及需要写什么内容。

如果写文档的目的不明确，那写出来的内容也不可能明确。有了一个明确的目的，不但能让我们有机会讨论是不是有必要写这个文档，而且写起来脑中也能有明确的内容。

有些时候，我们会遇到一些必须写的文档，但这些文档又不大可能有人来读。我们也要搞清楚这类文档是为谁写的，为什么要写。如果发现真的没必要写，可以跟委托人商量之后再定夺。

7.1.2　什么样的状态让人想写文档

我们前面考察了妨碍写文档的因素，那么什么样的情况能让我们更愿意写并且更轻松地写文档呢？

消除了妨碍因素，掌握写文档时的关键点后，以下几个“让人愿意写文档并能轻松写文档的条件”就自然而然地浮现了出来。

- 能在平时用的编辑器上写文档
- 能把文档分成几个文件来写
- 能用 Mercurial 等轻松实现版本管理
- 能集中精神编辑内容，不用顾虑装饰等外观问题
- API 参考手册与程序的管理一体化
- 平时的引用可通过 Web 浏览器共享
- 在提交文档时可转换成漂亮整齐的单一文件
- 写有用的文档

那么，满足这些条件的文档编辑环境有哪些呢？

如今市面上有很多文档编辑工具，它们各有各的特点，也各有各的长项与短项。这里我们从文档结构的观点出发，来了解一下这些特点的差异。

Wiki 是用来构筑半格（Semilattice）结构文档的工具。半格结构没有起点和终点，是一种各元素之间依靠引用链相连的网状结构。这种结构适合在某一主题下以不断添加关键字的形式编写文档片段。相反地，它不适合编写那种要从头到尾按顺序阅读并提交的文档。

与此相对的还有构筑树结构的工具。树结构存在起点，起点元素之下的各元素有着固定的从属关系。以图书为例，目录页下挂着各章的标题，各章下又挂着各节的标题。树结构的文档适合从头到尾按顺序阅读。不过，单纯的树结构在很多情况下都不大好用，所以我们会添加一个到任意元素的引用，将树结构改造成网状结构。虽然树结构看上去优于半格结构，但它必须有一个主干，所以不适合用来当没有固定结构的字典。

接下来我们将学习 Sphinx，它是构筑树结构文档的文档编辑工具。

Sphinx 是用 Python 编写的工具，用来构建多个以 reStructuredText 语法编写的文本文件，将它们转换为 HTML 或 PDF 等格式。Sphinx 可以将树的各个元素分割成多个文件进行管理。另外，这款工具的功能和特征满足刚才我们说过的"让人愿意写文档并能轻松写文档的条件"中的许多条，甚至能让人更加主动地想要写文档。

NOTE

当然，只要讲究方法、肯下功夫，不管什么工具都有可能让我们达到目的，但其过程是否给人带来压力就另当别论了。我们即将学习的 Sphinx 也完全不是那种在 GUI 中随便一操作就能写文章的工具，但它适合像写程序一样结构化地写文章。

接下来，我们将先来了解一下 Sphinx 的基本使用方法。然后根据本节列出的这些条件，分条学习如何用 Sphinx 实现它们。

7.2　Sphinx 的基础与安装

Sphinx 是一款文档编辑工具，具有十分全面的文档资料。

Sphinx 官方网站
　http://www.sphinx-doc.org/en/stable/
Sphinx 使用手册
　http://www.sphinx-doc.org/en/stable/contents.html

本节将对以下各项进行简要说明。

- Sphinx 的安装
- reStructuredText 入门
- 用 Sphinx 编写结构化文档的流程
- Sphinx 扩展

7.2.1 Sphinx 的安装

Sphinx 的安装流程并不复杂，但各位要尽量使用最新版本。这里建议各位使用最新的 Sphinx-1.3 系列（截至 2014 年 12 月时的最新版本）。

安装用以下命令执行。

```
$ pip install sphinx
```

这样，我们就装好了 Sphinx 关联的程序包，现在可以用 `sphinx-quickstart` 命令了。`sphinx-quickstart` 命令能自动生成启动 Sphinx 所需的数个文件。具体执行方法如 LIST 7.1 所示。

LIST 7.1 sphinx-quickstart

```
$ sphinx-quickstart -q -p SW-Project -a BeProud -v 1.0 sw-project
Creating file sw-project/conf.py.
Creating file sw-project/index.rst.
Creating file sw-project/Makefile.
Creating file sw-project/make.bat.

Finished: An initial directory structure has been created.

You should now populate your master file sw-project/index.rst and create other
 documentation
source files. Use the Makefile to build the docs, like so:
   make builder
where "builder" is one of the supported builders, e.g. html, latex or linkcheck.

$ ls sw-project/
_build  conf.py  index.rst  make.bat  Makefile  _static  _templates
```

执行完毕后，sw-project 目录下就生成了 Sphinx 项目，然后就可以通过 make html 构建 Sphinx 了。Sphinx 写文档的标准扩展名为“.rst”，所以这里会生成 index.rst 文件。

如果不加任何选项直接执行 `sphinx-quickstart`，则会以对话形式生成 Sphinx 项目。这种情况下，计算机将以对话形式提出几项询问，其中 Project Name、Author、Version 这 3 个

问题需要由我们来输入，其他问题则只需根据实际情况选择 Y 或 N 即可。在 LIST 7.2 的例子中，我们让文件生成在 sw-project 目录下。

LIST 7.2　sphinx-quickstart 对话模式

```
$ sphinx-quickstart
Welcome to the Sphinx 1.3 quickstart utility.

-（中间省略）-

> Root path for the documentation [.]: sw-project

-（中间省略）-

> Project name: SW-Project
> Author name(s): BeProud

-（中间省略）-

> Project version: 1.0

-（中间省略）-

$ ls sw-project
_build  conf.py  index.rst  make.bat  Makefile  _static  _templates
```

通过 sphinx-quickstart 设置的内容全都记录在 conf.py 文件中，所以日后想更改设置时需要编辑 conf.py 文件。另外，如果想用中文版的 Sphinx，可以在 conf.py 文件中作如下设置（LIST 7.3）。

LIST 7.3　conf.py

```
language = 'zh_CN'
```

详细安装流程以及 Sphinx 的初始设置请参考以下网站。

> Sphinx 入门
> http://www.sphinx-doc.org/en/stable/install.html

7.2.2　reStructuredText 入门

Sphinx 要用 reStructuredText（reST）语法写文档。这里我们学习几个具有代表性的 reST 写法（LIST 7.4）。

LIST 7.4　sample.rst

```
==============
章标题
==============

:项目 1: 字段列表内容 1
:项目 2: 字段列表内容 2
:项目 3: 字段列表内容 3

节标题 1
=====================

节内第一个段落的文章。
换行会被忽略。

第二个段落的文章。
用空行划分段落。
空行至少为一行，输入多少行效果都一样。

1. 规定编号的有序列表
2. 规定编号的有序列表

   #. 自动编号的有序列表
   #. 自动编号的有序列表

没有编号的无序列表按以下格式书写。

* 无序列表
* 无序列表

  + 低一级的无序列表
  + 低一级的无序列表

可以给词汇添加多种特殊意义。

- **强调**
- *斜体*
- ``显示为字符串``
- `链接字符串 A`_
- `链接字符串 B<http://docs.sphinx-users.jp>`__
- :doc:`index` 到对象文件的链接。会自动替换为章标题。

.. _链接字符串 A: http://sphinx-users.jp
```

为了从首页链接到这个 sample.rst 文件，我们在 index.rst 中作如下描述（LIST 7.5）。

LIST 7.5　index.rst

```
.. toctree::
   :maxdepth: 2

   sample
```

这样就可以构建 sample.rst 了。接下来我们执行 `make html`（LIST 7.6）。

LIST 7.6　make html

```
$ make html
```

构建结果将输出到“_build/html”目录下。用浏览器打开“_build/html/sample.html”可以看到如图 7.1 所示的输出结果。



图 7.1　Sphinx 输出示例

> reStructuredText 入门
> 　`http://www.sphinx-doc.org/en/stable/rest.html`

7.2.3　用 Sphinx 写结构化文档的流程

Sphinx 在很多地方都引入了结构化的概念，因此它为写文档的人提供了一个方便写文档且

方便整理的环境。下面就是利用 Sphinx 提供的结构化来编写文档的流程。

◉ 标题与元素

我们从零开始写文档时，往往会不知从何写起。虽然想到什么写什么不失为一种办法，但最好还是先列出标题大纲，整理一下文档内容的结构（LIST 7.7）。

LIST 7.7 先逐条列出标题

```
* 标题 1

  * 副标题 1
  * 副标题 2

* 标题 2
* 标题 3
```

然后再将想到的东西填进适当位置，使文章逐渐丰满起来（LIST 7.8）。

LIST 7.8 给标题添加内容

```
* 标题 1

  * 副标题 1
  * 副标题 2
  * 副标题 3

    在这里逐渐补充副标题 3 的内容。
    如果想到了与副标题 3 无关的内容，
    就补充到其他合适的地方。

* 标题 2

  * 副标题 1
  * 副标题 2

* 标题 3

  * 副标题 1
  * 副标题 2
```

这种具有明确父子关系或兄弟关系的文本称为结构化文本。本例中既用了标题和内容这种具有明显父子关系的逐条列记，又用了空格缩进的文本。在写文档时，如果我们脑中没有一个明确的框架，就需要不断重复这种堆砌“标题”与“表达内容的元素”（本例中是副标题）的工作。

◉ 单一文件内的结构化

用 reStructuredText 写的文章会被结构化，看上去条理分明。比如用 reStructuredText 描述会议记录并添加到邮件中，收件人就可以直接流畅地阅读。

结构化让各条信息之间的关系更加明确，并列、父子结构一目了然，而且信息的换位与重排也能很轻松地进行。

按照这种结构化规则写出的文本具有树结构（图 7.2）。

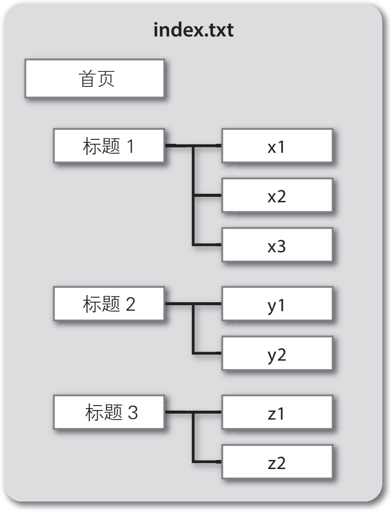


图 7.2　文本的树结构

◉ 文件与目录的结构化

在写文档的过程中，会遇到标题下的内容过于详细的情况，这会破坏文章的平衡性。另外，如果我们只往一个文件里写，文章的主干就会越来越模糊，事情也会越来越讲不清楚。当文章达到一定规模，其内容需要分类分割时，我们可以根据文件和目录将其分割并结构化。分割后，主干与分支之间必须保持一个父子关联。这样一来，文件就能在分割之后仍保持树结构了（图 7.3）。

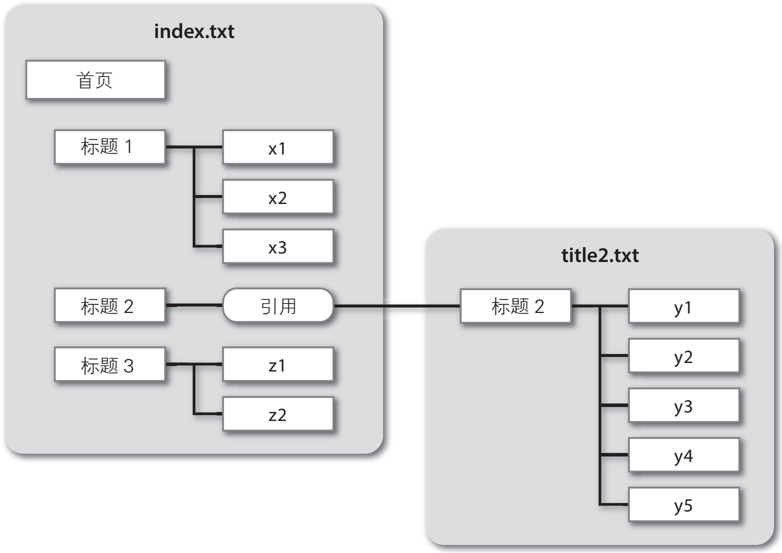


图 7.3 在维持树结构的前提下分割文件

等到文件数量多起来，可以将文件分组，按目录进行结构化。总而言之，就是灵活运用文件与目录，不断将结构化向前推进。

先将文档的章或节按目录或文件进行分割，这有助于我们今后对章节进行增减与重排。另外，由于文件被分成了多个，所以可以 1 人或多人同时编辑多个地方。举个例子，如果我们同时想到好几个点，那么可以同时打开多个文件做记录，这要比编辑单一文件更容易找到信息的位置，而且能很轻松地定位到之前中断工作的地方。

Sphinx 能将所有文档文件组合到一个树结构中。这样，所有文件都被排成了一个序列，并以让人能从上至下阅读的格式进行输出。该定义需在文档中用 `.. toctree::` 指令描述。

只要有了 toctree 这样一个主干，文档就能在被分割成多个文件之后仍保持其结构。

◉ 网状结构

只要按照一定的规则给文档加入关键字或到其他章节的跳转，就能实现 Wiki 那种灵活的网状结构。脚注、交叉引用、术语集、索引等就是此类网状结构（图 7.4）。

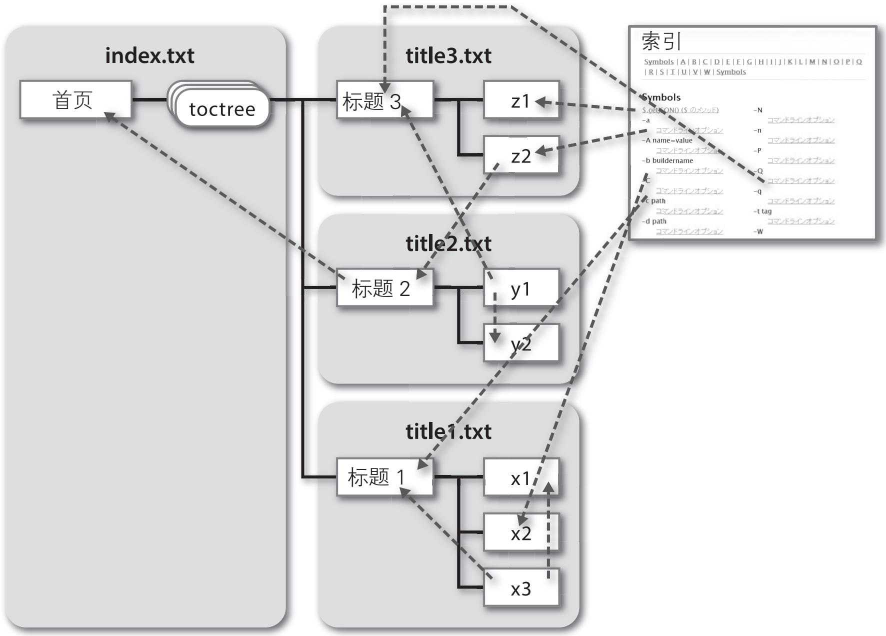


图 7.4　结构化文档的网状结构

链接可以让我们更容易地在文档中找到想找的信息。Sphinx 会为我们提供清晰的术语集、API 参考手册等信息，我们可以利用这个功能来统一术语。

比如在写文章的过程中发现有术语需要统一，我们可以先用 `:term:` 术语 `` 的形式将该术语写下来。由于 Sphinx 在 make 时会提示该术语没有对应的术语说明，所以我们完全可以把写术语说明的工作放到最后再做，这样既不会打断写文章的进度，也不必担心出现遗漏。

此外，我们还可以用 `` :doc:`../sub/index` `` 这样的形式指定引用页面的相对路径，Sphinx 在 make 时会自动将该页面的标题和链接填充到这里。

◉ 结构化的 3 个阶段

正如我们前面所介绍的，结构化分为如下 3 个阶段（图 7.5）。

① 单一文件内的结构化

② 分割成多个文件 / 目录并结构化

③ 连接成无直接父子关系结构的网络

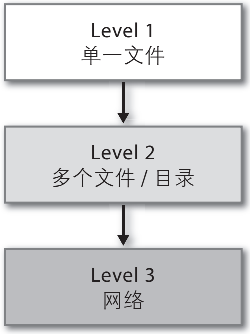


图 7.5　结构化的 3 个阶段

通过这样从内到外地让文章结构化、分组化，文档将自然而然地得到整理，写起来当然也会轻松许多。另外，Sphinx 以树结构为基础，与 Wiki 那种仅由网络形成的半格结构不同，阅读起来主次分明，易于理解。以树结构为主让文档整体有了一根脊梁，然后在各结构之间添加神经网络，这样可以加强文档阅读时的灵活性。

7.2.4　Sphinx 扩展

Sphinx 捆绑了 TeX 排版系统等扩展，另外也支持单独安装第三方开发的扩展。这些扩展包括流程图、序列图等图表的植入扩展、UML 画图、乐谱绘图、HTML 模板变更等。当然我们还可以自己开发扩展。

扩展功能在 Sphinx 的 conf.py 文件中设置。比如我们可以像 LIST 7.9 这样描述。

LIST 7.9　conf.py

```
extensions = ['sphinx.ext.pngmath', 'sphinx.ext.todo', 'sphinx.ext.autodoc',]
```

这样就激活了 Sphinx 主体程序自带的 3 个扩展功能。这里将介绍的是 sphinx.ext.pngmath。

pngmath 扩展用来在文档中以图片形式显示数学公式。我们只要在文档中描述了代表数学公式的字符串，该扩展就会在构建时将它们转换为 PNG 图像文件植入文档。不过，用这个扩展的前提是具备 LaTeX 环境①。

我们在文档中作如下描述（LIST 7.10）。

LIST 7.10　math.rst

```
=======
数学公式
=======
```

① Sphinx-users.jp 网站上有介绍 LaTeX 环境的安装流程。——编者注

```
.. math:: (a + b)^2 = a^2 + 2ab + b^2
```

构建之后会显示如图 7.6 所示的数学公式。

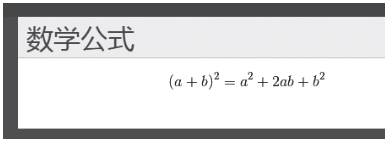


图 7.6　sphinx.ext.pngmath 的数学公式绘图

Sphinx 对数学公式的支持

http://www.sphinx-doc.org/en/stable/ext/math.html

经由 LaTeX 输出 PDF 文档[①]（日语）

http://sphinx-users.jp/cookbook/pdf/latex.html

7.3　导入 Sphinx 可解决的问题与新出现的问题

本章最开始就列举了下列“让人愿意写文档并能轻松写文档的条件”。

- 能在平时用的编辑器上写文档
- 能把文档分成几个文件来写
- 能用 Mercurial 等轻松实现版本管理
- 能集中精神编辑内容，不用顾虑装饰等外观问题
- API 参考手册与程序的管理一体化
- 平时的引用可通过 Web 浏览器共享
- 在提交文档时可转换成漂亮整齐的单一文件格式
- 写有用的文档

现在我们分条学习如何用 Sphinx 实现这些条件。

① 该部分内容是 Sphinx-users.jp 网站针对日语文档的生成这种情况而编写的。英语文档的生成则参照 Sphinx 手册即可。——编者注

7.3.1 由于是纯文本，所以能在平时用的编辑器上写文档

想必程序员都有过读纯文本文档的经历。特别是开源程序的附属文档，其中有许多图表都是由纯文本表达，它们和程序一起被视为源码进行管理。为什么很多人选择用这种方法写文档呢？理由有以下几点。

- 不必借助特殊工具，仅用纯文本就能表达结构或图表
- 编写、阅览文档不依赖特殊工具，无论该工具有偿无偿
- 可以用源码管理工具来管理文本文件的变更
- 写代码和写文档都能在惯用的编辑器上进行

可见，用纯文本写文档并不是心血来潮，它是有明确优势的。

Sphinx 就是用纯文本写文档。用户按照特定的规则描述纯文本，然后 Sphinx 来解释该规则，将文本以 HTML、PDF 等格式输出，这就是 Sphinx 的作用。

图片要以独立文件的形式统一保存，跟文档的文本文件放在同一目录或其子目录中（LIST 7.11）。这样，一旦日后遇到容量等问题需要改变大小或格式时，能方便地统一转换。

LIST 7.11 用于植入图片的 figure 指令

```
方便多人同时编辑，也便于用版本管理工具进行管理

.. figure:: images/7-sphinx-vcs-manage.png

   用 Mercurial 管理 Sphinx 文档
```

7.3.2 信息与视图相分离，所以能集中精神编辑内容，不用顾虑装饰等外观问题

Sphinx 的以下特征实现了信息（Data）与视图（View）的分离，所以我们能集中精神写文档。

- 用 reStructuredText 写文档看不到最终的显示效果
- 可以集中精力描述逻辑结构化的正文
- 最终的设计由 Sphinx（或者由制定主题的设计师）调整
- 相当于其他文档编辑工具的大纲功能

前面我们学习了如何用 reStructuredText 写文档以及如何用 Sphinx 构建并输出。总而言之，我们可以把装饰和外观交给 Sphinx 处理，自己集中精神写文章内容。

比如，我们要用 Sphinx 编写如下会议记录（LIST 7.12）。

LIST 7.12　会议记录示例

```
============================
2015/2/17 会议记录
============================
:参加者： SW 商事平野、BP 清水川、BP 小田切、BP 冈野
:日期及时间： 2015/2/17（二）10:00 ~ 11:30
:场地： SW 商事的会议室

今日议程
==================
1. 介绍新成员
2. 面向开发者的程序库的开发情况
3. 11 月起的工作方针

介绍新成员
=================
冈野：
   Python 经验 10 年。平时用 Django 框架和 GoogleAppEngine 进行开发。
   参与了许多面向 Python/Django 的开源程序库的开发，
   其中以面向移动电话的开发支持库 django-bpmobile 为代表。
```

这篇会议记录是用 reStructuredText 写的。如果用 Sphinx 构建，可以获取如图 7.7 ～图 7.9 所示的经过整理的 HTML 或 PDF 文件。



图 7.7　Sphinx 的 HTML 输出（default 主题）



图 7.8　Sphinx 的 HTML 输出 (bizstyle 主题)

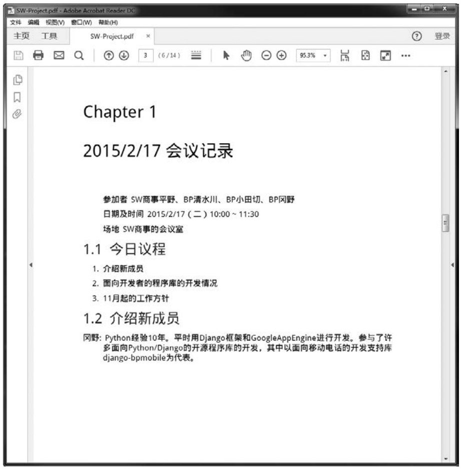


图 7.9　Sphinx 的 PDF 输出 (latexpdfja)

此外，Sphinx 还搭载了强大的代码高亮功能，它可以给程序代码等自动搭配颜色，提高可读性。

代码高亮功能由 Sphinx 捆绑的 Pygments[①] 提供。支持的格式方面，编程语言、设置文件、HTML 模板等加起来有 200 种左右，而且仍在增加 (图 7.10 ~图 7.12)。

① http://pygments.org/docs/lexers/



图 7.10　Windows ini 格式的高亮

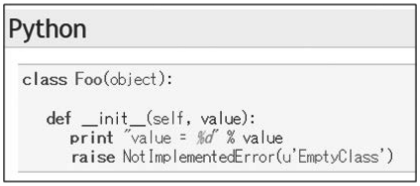


图 7.11　Python 代码的高亮

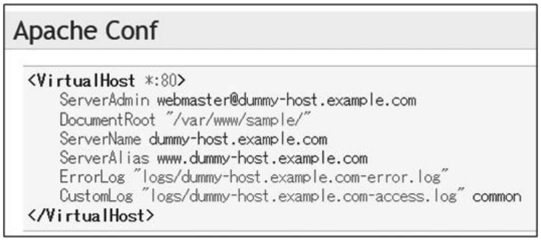


图 7.12　Apache conf 格式的高亮

7.3.3　可根据一个源码输出 PDF 等多种格式

前面我们了解了纯文本的好处以及文件分割的好处。不过，我们仍然希望在印刷和交付时，文档能被整合成一个文件。

另外，虽然文本文件形式的文档与 PDF 文档有着同样的信息价值，但文本文件形式的文档总会给人一种廉价感。为避免因此而受到影响，我们在上交文档时最好准备一份经过简单排版和装饰的 PDF 文件。

Sphinx 可以将同一份源码文件转换成多种不同的格式。输出格式支持 HTML、PDF、EPUB、man、LaTeX、HTML Help（chm）等（图 7.13）。另外，导入自制的扩展后，还可以支

持一些原本不支持的格式。

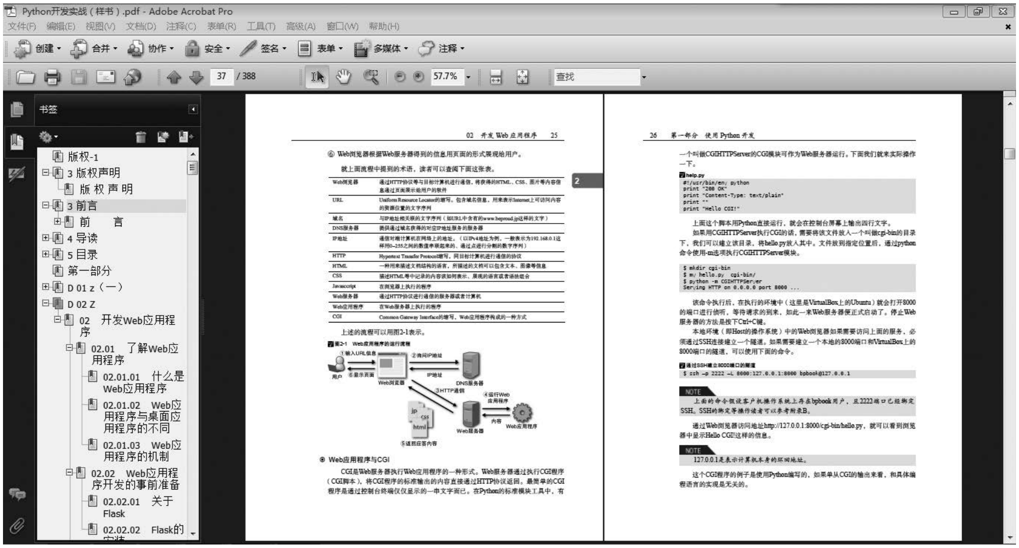


图 7.13　Sphinx 输出的 PDF

7.3.4　通过结构化，文档可分成几个文件来写

程序员在写程序时不会将所有代码都写在同一个文件里。因为历史和长期以来的经验告诉我们，这样做是非常不明智的。源码要以概念或目的为单位分割成多个文件，在目录中进行分类管理。

这个做法对于文档同样有效。分割、分类的标准可由写文档的人随意制定，但其中还是有一些技巧可言的。比如 LIST 7.13 这个例子就能让人一眼看出文档的结构，非常清晰。

LIST 7.13　文档目录结构示例

```
文档 /
  +-- index.txt
  +-- 项目管理 /
  |    +-- index.txt
  |    +-- project-goal.txt
  |    +-- member-and-structure.txt
  |    +-- phase1-schedule.txt
  +-- 设计 /
  |    +-- index.txt
  |    +-- middleware-versions.txt
  |    +-- framework-comparision.txt
```

```
|    +-- server-structures.txt
|    +-- components-and-interfaces.txt
+-- 开发 /
|    +-- index.txt
|    +-- repository.txt
|    +-- coding-rule.txt
|    +-- develop-environment.txt
|    +-- release-packaging.txt
+-- 参考手册 /
|    +-- index.txt
|    +-- install-and-setup.txt
|    +-- class-and-functions.txt
|    +-- api-references.txt
+-- 会议记录 /
     +-- index.txt
     +-- 20150201-meeting.txt
     +-- 20150208-meeting.txt
     +-- 20150215-meeting.txt
```

这样分类之后，我们写文档时的注意力就会转移到以下几项上。相较于在单一文件中编写文档，这样能更加轻松地整理文档内容。

- 根据分类或文件名调整该文件涉及内容的范围
- 文件内容或文章量达到一定规模后，重新整理内容、分割文件
- 文件达到一定数量后，将同一分类的文件归入一个子目录

如上面例子所示，Sphinx 可以借助文件分割以及目录分类来构筑文档。这些分割开的文件会在使用 Sphinx 构建时通过链接等方式添加关联，并采用适合 HTML、PDF 等格式的形式输出。

至于输出之后的效果，完全可以交给 HTML 或 PDF 等输出算法来处理。写文档的人只需关心文档结构是否符合逻辑，链接数量是否恰到好处等。也就是说，我们能完全以逻辑思考为中心来编辑文档，不必去想多余的事。

7.3.5　能用 Mercurial 等轻松实现版本管理

Sphinx 的文档由多个目录下的多个纯文本文件共同构成。图片文件也和文本文件一样保存在目录中。在这种结构下，我们能以极细的粒度进行版本管理，而且即便出现多人同时编辑文档的情况，各个变更之间也不会发生冲突（图 7.14）。



图 7.14　用 Mercurial 管理 Sphinx 文档

7.3.6　API 参考手册与程序的管理一体化

作为开发者，我们有时候会自己写 API，有时候又会拿现成的 API 来用。理想的情况是写好 API 的同时就能做好 API 参考手册，但如果是没有人读的东西，做出来也只是白费功夫。所以什么时候该做什么东西，要仔细结合重要性进行考量。

进行 Python 开发时，只要 docstring 用得好，完全可以解决这一问题。docstring 指的是写在 Python 模块、类、函数最开头的字符串对象。比如，我们会用 docstring 在函数的开头像 LIST 7.14 这样描述文档。

LIST 7.14　path.py

```
# -*- coding: utf-8 -*-

def commonprefix(path_list):
    """
    返回路径的“`path_list`”中最长的公共前缀
    （依次判断路径名的每一个字符）

        >>> commonprefix(['/usr/bin/python', '/usr/local/bin/python'])
        '/usr/'
        >>> commonprefix(['/usr/bin/python'])
        '/usr/bin/python'

    如果“`path_list`”为空，则返回空字符串“''”

        >>> commonprefix([])
        ''
```

```
    请注意，由于每次只判断一个字符，
    所以可能返回非法路径

        >>> commonprefix(['/usr/local/bin/python', '/usr/local/bin/pylint'])
        '/usr/local/bin/py'
    """
    if not path_list: return ''
    s1 = min(path_list)
    s2 = max(path_list)
    for i, c in enumerate(s1):
        if c != s2[i]:
            return s1[:i]
    return s1
```

如 LIST 7.15 所示，以这种方式写进去的函数文档可以在交互模式中引用。

LIST 7.15　函数文档的引用

```
>>> help(path.commonprefix)
Help on function commonprefix in module path:

commonprefix(path_list):
    返回路径的“`path_list`”中最长的公共前缀
   （依次判断路径名的每一个字符）

        >>> commonprefix(['/usr/bin/python', '/usr/local/bin/python'])
        '/usr/'
        >>> commonprefix(['/usr/bin/python'])
        '/usr/bin/python'

-（中间省略）-
```

Python 还拥有 doctest 功能，它能够识别出在 docstring 中描述的，也就是和交互模式显示信息相同的部分并进行测试。进行 doctest 的最简单的方法就是执行 LIST 7.16 中的命令。

LIST 7.16　doctest

```
$ python -m doctest -v path.py
...
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

这样一来，函数文档里写的内容和函数的功能就不会再有偏差了。另外，我们可以放心地

将 docstring 植入 Sphinx 文档。植入时要用到 sphinx.ext.autodoc 扩展，具体描述如下（LIST 7.17、LIST 7.18）。

LIST 7.17　conf.py

```
sys.path.insert(0, '【path.py 所在目录的路径】')
extensions = ['sphinx.ext.autodoc',]
```

LIST 7.18　path.rst

```
================
AutoDoc 范例
================

.. autofunction:: path.commonprefix
```

然后我们就能够得到图 7.15 中的输出结果了。



图 7.15　用 sphinx.ext.autodoc 将 docstring 植入 Sphinx 文档

可见，只要 API 的实现及其相关文档保持在可测试状态，并保证其能自动植入 Sphinx 文档之中，许多问题就迎刃而解了。另外，如测试驱动开发一样，将介绍 API 使用方法的文档兼测试写在实现 API 之前，这既能让人们对该 API 的使用有一个更明确的认识，也能给开发带来帮助。

7.3.7　通过 Web 浏览器共享

文档内常常包含开发规则或 API 参考手册等内容。让成员能在 Web 上浏览到这些信息会带来许多好处。

当我们想和其他开发者共享某个文档，进而展开讨论或交流时，如果能通过 URL 指定该文档，那将让共享变得非常方便。而且这样做不需要在邮件上挂附件，能避免出现多余的文档副本。

Sphinx 用起来最方便的输出格式就是 HTML。我们可以设置一个机制，在 Mercurial 的版本

管理之下提交 Sphinx 源码时，用 Jenkins 自动将其构建成 HTML，这样一来就能随时在 Web 上看到最新的文档了。与 Jenkins 的关联我们将在第 10 章中详细了解。

7.3.8　导入 Sphinx 后仍存在的问题

前面我们了解了 Sphinx 作为文档编辑工具的一面，但有些问题并不是导入 Sphinx 就能解决的。另外，对于非程序员或非技术人员来说，Sphinx 的某些优点反而会变成缺点。

所以，我们这里要探讨导入 Sphinx 无法解决的问题以及导入后新增的问题。

◉ 写有用的文档

我们在“让人愿意写文档并能轻松写文档的条件”中提到了“写有用的文档”这一条。可惜的是，仅导入 Sphinx 并不能解决这一问题。在 7.4 节，我们将学习如何写有用的文档。

◉ 审查需要多花心思

许多文字处理软件都配备了审查或审校的功能，但 Sphinx 中并没有这类功能。所以在进行审查校对、反馈校对内容等工作时，需要额外花些心思才行。

Sphinx 中有用来记录 todo 的扩展表示法。使用这个功能前，先要在 conf.py 中作如下设置（LIST 7.19）。

LIST 7.19　conf.py

```
extensions = ['sphinx.ext.todo',]
todo_include_todos = True
```

然后像 LIST 7.20 这样在文档中描述审查校对的内容。

LIST 7.20　在 todo 指令中描述校对内容

```
今日议程
===================
1. 介绍新成员
2. 面向开发者的程序库的开发情况
3. 11 月起的工作方针

.. todo::（sato）缺少 ** 事务联络 **，请添加。
```

如果想获取 todo 的一览表，需要在文档的任意位置加一行 `..todolist::` 然后 make html。

> sphinx.ext.todo - Support for todo items
> `http://www.sphinx-doc.org/en/stable/ext/todo.html`

◉ 部分人只会在 WYSIWYG 编辑器上写文档

前面也说了，Sphinx 的信息与视图是分离的，所以我们能集中精力去编写文章，不必为装饰等外观方面的问题分心。但是对于一部分人而言，不能把握输出效果就写不下去文章。这一类人会觉得没有 WYSIWYG 编辑器的 Sphinx 很难用。这个问题暂时还没有很好的解决方法。

◉ 输出 PDF 需要搭建相应的环境

虽然 Sphinx 支持多种输出格式，但仅有它本身时，是不能输出 PDF 的。输出 PDF 有两种方法（经由 LaTeX 和使用 reportlab 扩展）。这两种方法各有特点，其中经由 LaTeX 的输出更加直观。

关于用 Sphinx 输出 PDF 的方法，`http://www.sphinx-doc.org/en/stable/index.html` 网站上介绍了相关的导入流程以及使用方法。此外，该网站上还介绍了将 Sphinx 文档经由 LaTeX 输出成 PDF 的方法。

> 用 Sphinx 生成 PDF 文件
> `http://www.sphinx-doc.org/en/stable/tutorial.html?highlight=pdf`

7.4 文档集的创建与使用

各位在写文档时都注意了哪些点呢？另外，要想写出一篇有用的文档，有哪些点需要注意呢？

文档是有读者的。如果在写文档时没有考虑读者，那我们的意识就会偏到“应该加入哪些信息”“应该写到哪种程度”上，结果就是相关信息越添越多，最后却忘了原本要表达的信息，导致文档不能达到原定的目的。能让读者觉得是好文档的文档，必然考虑了“读者是谁”以及“怎样写能让读者看得更明白”等问题，确保为读者提供了充足且有用的内容。

文档的目标读者、内容、深度等问题，可以拿过去项目的文档来参考。我们知道，“文档的写法”“版本库的用法”“项目的推进放法”这类东西都是可以重复利用的。同样道理，一个项目文档的许多元素完全可以拿到另一个文档中重复利用。

7.4.1 什么是文档集

如今有一种思路，就是为了重复利用文档而将文档模板化，然后构建成文档集（Documentation Portfolio）。《Python 高级编程》[①] 一书将文档集称为文档工件集，并作出如下定义。

---

① 原书名为 *Expert Python Programming*，Tarek Ziadé 著，姚军等译，人民邮电出版社，2010 年 1 月。
——译者注

从作者的角度，这可以通过拥有一组可复用的模板和描述如何、何时在项目中使用这些模板的指南来完成，它被称为文档工件集。

也就是说，一个文档集由以下内容构成。

- 文档模板集（规则、流程、会议记录、设计等）
- 使用文档模板的导航（何时用、怎样用等说明）

7.4.2　项目所需文档的一览表

Scoot Ambler 著的《敏捷建模：极限编程和统一过程的有效实践》一书中有文档集的相关信息，可供各位参考。他在书中提到了项目所需的文档一览表，同时举了例子，现整理如下。

- 文档一览表（精选）
  - 协议模型：关于系统的技术性界面
  - 设计上的已确定事项：设计和架构方面的重要决定的记录
  - 对上级的概要说明：系统构想、需求、当前的预估、风险、人员计划、日程安排
  - 使用文档：使用时所需信息、流程、环境的概述
  - 项目概要：开发时所需信息、流程、环境的概述
  - 需求文书：定义系统需要完成的目标
  - 支持文档：给技术支持者的培训资料、问题排查、维护团队的联络方式一览表等
  - 系统文档：系统概要、构架、需求事项等的概要
  - 用户文档：使用说明书、培训资料等

敏捷建模的文档列表基本网罗了所有必备信息。但这仍存在一些问题，比如从完全没有文档的状态起步时，我们很难一次性将这些文档全写出来。所以第一步，可以先处理自身团队或组织所需的部分，或者是以过去的文档为基础，将文档的目的抽选出来进行类似上述分类，进而模板化。最终我们将构筑出适用于自己或团队的文档集，供我们在今后的项目中加以利用。

接下来，我们将根据目标读者的类型分别了解一下文档集。

7.4.3　面向项目组长、经理

普通程序员很难想象项目组长、经理们想要的信息，所以更谈不上将它们写入文档。这里最好的办法就是向项目组长、经理询问具体想要的信息。各位不妨以本节所讲的内容为基础，跟项目组长、经理探讨一下访问客户之前应该掌握哪些信息，应该将哪些东西落实在文档中。

◉ 项目的目标（终点）

在项目目标中，我们要写出“用户导入我们即将开发的系统后能获取何种收益”“与以往相比有哪些改善”之类的信息，而技术和系统方面的目标则不必在此太过重视。当然，有些项目本身就是为了开发 Python 程序库，这时技术和系统方面的目标就成了必须写入目标的事项。不过，这种情况也必须写清楚为什么要开发这个东西。

项目的目标（终点）自然应该跟所有参与项目的成员共享，但意外的是，这一点在现实中贯彻得并不好。即便团队中某些成员只负责敲代码，我们也应该把目的共享给整个团队。更何况，在文档中写明目标对项目组长本身也有帮助。或许尚不熟悉项目运营的组长会觉得“目标会经常跟着客户的要求变”，但实际看来，目标真正改变的项目并不多。我们不光要询问客户的要求，还要将目标落实到字面上，然后跟客户以及项目组成员共享出来，这样才能给项目创造出一个主干，使开发有条不紊。如果这一点上搞不清楚，那这个项目从头至尾都不会稳定，难免发生差错，甚至会导致返工。

◉ 体制

在项目的体制中，要写明各个参与者（团队）的职务，使成员们清楚自己在项目中的定位。职务不能只写名称，还要写明各职务需要做的工作，让人明白每个职务该负起哪些责任。如果只写一个“主程序员”，那么任谁都搞不清楚这个职务应该负责什么。这部分虽然不必写得很严密，但必须让每个职务对应的人员清楚自己的职责范围，不然项目开发的过程中必定出现大问题。另外，某些职务或团队的人数需要根据参与时期进行调整，这类信息最好也写进来。

◉ 需求

需求是很重要的。在项目开始之前，需求就已经朦胧地存在于客户脑中了。我们的设计与开发全都要以需求为准。在开发过程中，需求是考察系统完成度的指标，如果需求不明确，我们完全无法判断系统的发展方向是否正确，也无法考察开发到了哪里。到了运营阶段，我们也需要参考需求来了解系统的概念或衡量业务变更的影响。

在定义需求时，一般要将需要哪些功能、需要怎样运营、需要运行性能达到何种程度等信息逐条列出或者汇总成表。

◉ 日程、咨询项目表

对实现需求的日程进行描述。能左右日程的因素有很多，比如客户的要求、当前可处理的范围、能影响到日程的外部因素，等等。在初期阶段，日程常会受各种因素影响而变化，所以要定期重新审视并做好记录。

除此之外还有咨询项目表，用来与客户协商尚不明确的事宜。

7.4.4　面向设计者

面向设计者的文档需要对构架和概念多加描述，但随着时间推移，选择该设计、基础结构、开发语言的理由会显得越来越重要，所以这些也要写在文档中。另外，在 OS 和开发语言方面，最好也写上选择该版本的理由。以 Python 为例，假设我们选用了较旧的 2.5 版，理由可能是只有这个版本提供了我们用作基础结构的服务，也可能是当时没有发布新版本，或者是 OS 的程序包管理中没有新版本，又或者是公司的方针规定不采用最初的主版本，总之可能的情况非常之多。如果文档中没有对这些情况做记录，一旦将来要探讨版本变更问题，那将会是一件非常麻烦的事。

涉及下列各项时，请各位务必记录选择的理由。

- OS、语言、版本的选择
- 构架设计
- 基础设施设计

7.4.5　面向开发者

在搭建开发环境方面，我们提倡导入自动化机制来尽量缩短流程，但还是难免会遇到无法自动化的部分，或者是自动化成本过高的情况。因此，我们应该将流程尽量详细地记录下来，以备不时之需。有些时候，流程文档的记录会不够全面，比如只写了搭建步骤却没写构建选项，等等。要知道，流程文档是写给不懂这些的人看的，所以应该将命令行要输入的内容全部网罗进去。

NOTE

能自动化的地方不能太吝啬成本，应当尽量自动化，从而降低搭建环境和编辑文档的成本。

在一些环境搭建流程尚未优化的项目中，单是给一个开发者搭建环境就可能花费超过一周的时间。这种情况下，每当遇到实现或测试阶段都会消耗大量的人力资源。软件开发项目中的许多机制都可以自动化，其中与搭建环境相关的自动化最好在项目起步时就准备好。

本书将在第 9 章和第 11 章中介绍自动化搭建环境的相关内容。

7.4.6　面向客户

面向客户的文档包括指导如何使用成品程序的学习手册，以及总结了日常操作流程的使用指南等。

当然，前面讲到的设计、开发、搭建环境等的文档一般也会交付给客户，但除了客户想自己修改程序之外，这些文档的内容都太过晦涩，而且客户无法对内容加以干涉。与此相对，使用手册是客户日常会用到的东西，而且客户在给这类文档的内容提要求时更清楚自己想要什么。

是否需要面向客户的文档，或者文档中该有哪些内容，这都需要与客户认真探讨系统的使用情境之后再决定。如果定不下来，那么很可能是客户自己对系统没有一个明确的设想。这种情况下，客户可能在收到成品后或开始使用后才发现“这跟我想要的不一样”。所以为了防止出现这类问题，我们必须与客户认真探讨，共享成品在开始运营后的情境，编写出所需的文档。

7.5 小结

本章对“什么样的环境便于开发者写文档”进行了整理，并以 Sphinx 为例介绍了实现此类环境的方法。

我们无法保证大程序从一开始就能被正确实现，同样道理，我们也无法保证大文档从一开始就能有一个正确的结构。Sphinx 这类结构便于文档阶段性成长，它能防止我们在写文档上浪费劳动力，同时也能作为指向标，引导我们最终完成一份恰到好处的文档。

另外，对于“写有用的文档”这一无法用工具解决的问题，各位可以运用文档集来弥补。将已有文档的结构和构造升华为文档集来重复利用，不但能逐渐提高文档的品质，还能有效降低文档编写的成本。

第 8 章　模块分割设计与单元测试

Python 上手容易，编程者可以很快投入到应用的开发之中。但这里不能操之过急。在动手之前，各位要先问一问自己：理解接下来要做的东西了吗？知道该怎么做了吗？

没错，敲代码前还有一个必经的步骤，那就是设计。

另外，敲完代码之后还少不了测试。如今，在极限编程等敏捷开发方法的影响下，人们对测试的认识正在从“麻烦”逐渐向着“主导开发的步骤”转变。

从某种意义上讲，设计是对测试的一种输入，因为我们只能在测试的过程中检验自己明确设计出来的东西。另外，易于测试的设计还能让源码维护省力不少。为了让测试能够立竿见影，我们选择将功能分割到模块之中进行开发。

本章会向各位介绍将功能分割到各个模块的设计方法、测试手法以及如何根据测试结果改善设计。

8.1　模块分割设计

应用是由功能集合而成的。功能则要通过函数、对象等的相互作用来实现。在 Python 中，函数、类等（以下统称为组件）整合在模块里，模块又整合在程序包里。设计的第一步是设计功能，然后才轮到实现功能的各个组件。

8.1.1　功能设计

在功能设计阶段，我们要敲定即将开发的应用包含哪些功能，明确各功能的输入输出。

从应用的角度看，输入输出不仅包括用户能看到的这部分，还涵盖了与外部相关系统的接口、向数据库保存的数据等。

首先我们要给功能写出一个方案。所谓方案，就是描述用户在使用功能时会与系统进行何种交互的文本。我们要通过这一步明确用户向系统输入的内容，以及系统该向用户显示的内容。

将方案所示的整个流程画成图，即为页面迁移图。这里要写下各个页面的功能。作为一款 Web 应用，应当包含下面几项。

- URL
- HTTP 方法
- 安全（登录、权限等）

- 输入
  - 用户输入
  - 从数据库或文件读取
  - 外部系统发来的内容
- 输出
  - 页面输出
  - 向数据库或文件写入
  - 外部系统的调用

另外，要明确输入和输出的对应关系。连接输入与输出是功能的职责。在功能测试中，我们要查看的也是这些输入和输出。

通过功能设计，我们确定了即将开发的应用应当怎样运作。做完了这一步，接下来就是设计“如何实现这些功能”。简单的功能通常可以一步到位，但在实际开发中，绝大部分功能要么复杂，要么必须与多个功能协同工作。对付复杂的功能时切忌强求一步到位，最好将其视为多个部分（组件）的组合。将复杂功能分割成组件的好处在于可以重复使用同一组件来完成相同的工作，从而有效避免应用内的矛盾。

下面，我们将对构成 Web 应用的组件进行了解。随后再来学习如何将功能分解成组件。

8.1.2 构成 Web 应用的组件

将功能分解成组件需要用到一个指标，我们将这个指标称为软件架构。

Web 应用架构中最有名的当属 MVC（Model View Controller，模型 – 视图 – 控制器）了。MVC 根据职责不同对 Web 应用的互动（应用与用户关联的）部分进行了分割。M 是 Model，它负责功能的主要逻辑与数据；V 是 View，负责将 Model 以用户能理解的形式显示出来；C 是 Controller，它的工作是根据用户的输入将 Model 和 View 联系起来。

近来的 Web 应用框架已经基本上集成了 Controller 的功能。与此同时，View 则更多地被分割成 HTML 模板和显示逻辑两部分。比如，Django、Pyramid 等框架就很少需要明确写出 Controller，绝大部分情况都是 Model、View、Template 结构。

Model 部分在 MVC 中很少被提及。不过，当应用不是单纯的 Web+DB 形式时，其必然会与外部系统存在某种协作。另外，虽然绝大部分 Model 会被永久保存，但不可否认有些 Model 只是为了构成功能而存在的。鉴于这些因素，我们把 Model 分成 ApplicationModel 和 DomainModel 两类来考虑。DomainModel 是拥有持久（保存在文件或数据库中）状态的模型，而 ApplicationModel 是用于构成功能的模型，虽然它们也能具有状态，但这些状态通常不会被永久化。

现在来总结一下构成 Web 应用的组件。由于 Controller 可以完全交给框架来处理，所以构成功能的组件应如图 8.1 所示。

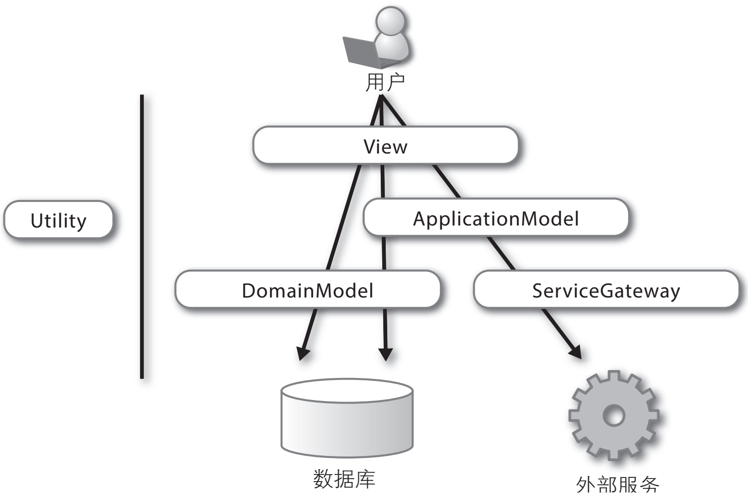


图 8.1　Web 应用构架

- View

  它的工作就是接受请求然后作出响应。输入主要为请求的传值参数，输出则是响应体。它会检查用户输入的数据，另外还会加载模型以及调用ApplicationModel。最后，它会将结果对象转换成可以显示给用户的HTML，生成响应体。

- DomainModel

  拥有持久状态的对象。大部分情况下，对象的状态保存在RDBMS中。Python上能使用的O/R映射工具（Object-Relational Mapper）以SQLAlchemy最为有名。另外，它还具有能改变自身状态的业务逻辑（Business Logic）。

- ApplicationModel

  没有持久性，其状态是暂时的。状态大多只能维持Web应用的一次请求，或者一定程度上完整的多个页面迁移期间。它们通过一般的类、模块中的函数来实现。经View调用后，执行其对应的各个DomainModel内定义的方法。它们大多不进行具体的处理，只负责传递处理的流程。也正是因为这个，它们依存的模块通常较多。

- ServiceGateway

  表示外部服务。其作用是为了让Web API的简单封装、邮件发送、启动其他处理等直接依存于外部系统的内容要能够不包含在Model和View之中。

- Utility

  它们大多都很小，不具备状态，输入只有函数传值参数，输出只有返回值。当然，它也可以拥有多种形式的输入，以对象状态维持自身直到最终调用方法，但输入输出要符合前面所说的要求，在该对象内完成所有处理。

一个功能对应一个 View。

ApplicationMode 和 DomainModel 被 View 调用。如果 DomainModel 只有一种，那么程序将通过 View 直接调用 DomainModel。当同时关联着多个 DomainModel 时，程序则要通过 View 调用 ApplicationModel。

ServiceGateway 的作用就是一个简单的封装，它里面不能包含应用固有的内容。ApplicationModel 则要使用 ServiceGateway 进行应用固有的处理。

> **NOTE**
>
> SQLAlchemy
>
> 它是连结对象与 RDBMS 的 O/R 映射工具库。虽然 SQLAlchemy 并不是 Python 唯一的映射工具（其他还有 SQLObject、Storm 等），但相比之下它的功能最为丰富，文档也相对齐全。像 Pyramid、Flask 等非全栈式（不具备 O/R 映射工具）框架的说明大多以使用 SQLAlchemy 为前提，这在如今的 Web+DB 应用开发中已经逐渐成为了一条不成文的规定。

8.1.3 组件设计

接下来我们把功能分割到各个组件当中。

◉ 根据输入设计 View

首先对付输入。作为一款 Web 应用，输入的主要来源应该是用户输入的数据。除此之外，已存在的数据和从外部系统获取的数据都属于输入。

用户输入的数据由 View 接收。另外，View 还负责向用户传递数据。交给模板的内容要由 View 进行输出。在 View 的内部会调用 ApplicationModel 或 DomainModel，这些 Model 也是输入之一。还有，由于 View 没有状态可言，所以大多数情况以函数（以下称 View 函数）形式实现。

◉ 初步设计与 View 关联的 ApplicationModel

对 View 函数而言，函数传值参数是输入，返回值是输出。另外，它只关联一种 Model。

这一阶段我们先不考虑 DomainModel 的相关问题，先给每个 View 函数定义一个

ApplicationModel。View 函数和 ApplicationModel 建议根据功能名来命名，比如“{ 功能名 }_view、{ 功能名 }Model”。除此之外，还要给 ApplicationModel 预先定义一个方法，名称同样根据功能名来起。

◉ DomainModel 的设计

接下来定义 DomainModel（ View 和 DomainModel 的定义可以并行进行。实际上，在设计 View 之前就可以一定程度上对 DomainModel 进行定义了)。由于 DomainModel 有状态，所以要用类来定义。

定义好 DomainModel 的类之后就要考虑方法了。要明确各个方法是如何改变 DomainModel 的状态的。这些状态要定义成属性。另外，如果要通过调用方法来改变其他相关 DomainModel 的状态，需要在目标 DomainModel 里定义方法，然后调用目标 DomainModel 里的方法。切记不可以直接改变其他 DomainModel 的状态。

◉ 关联 ApplicationModel 和 DomainModel

接下来给 ApplicationModel 和 DomainModel 添加关联。在 ApplicationModel 的方法中定义需要调用的 DomainModel。然后检查功能的输入输出是否满足要求。

◉ 整理 ApplicationModel

当 ApplicationModel 只关联着一个 DomainModel 时，应当删除该 ApplicationModel，让 View 直接调用 DomainModel。在 ApplicationModel 中，对象 Domain 模型一致地要整合成一个类。整合好之后需要立刻给类命名。如果无法确定类名，就尽量不要整合。

◉ 从组件设计转入实现

到这里，我们已经将功能分割到了各个组件之中。接下来就是把它们向 Python 的模块、程序包一步步整合了。

8.1.4　模块与程序包

现在我们根据目的把组件整合成模块或程序包。开发 Web 应用时，要每一个功能制作一个程序包，功能内的各个组件（ View、Model 等）分别制成模块。在 Python 中，各个组件由类和函数来实现。

NOTE

我们来整理一下 Python 的模块与程序包。

模块

Python 的源码文件就是一个个模块。里面整合着多个类、函数、变量等内容。虽然我们可以把处理写到模块内，但当我们开发的应用由多个模块构成时，最好只让它作为一种函数和类的集合体出现。

程序包

整合在一起的多个模块。它就是把程序包内包含的模块文件与 __init__.py 整合到了一个单独的目录下。

命名空间程序包

程序包的一种比较特殊的形式，用来向多个位置安装相同名称的程序包。因为只有在分配库文件的时候才会用到它，所以一般应用都与它无缘。

◉ 项目的文件结构

以 MVC 架构为基准时，项目的实际文件结构如下所示。

```
project/
 +-- __init__.py
 +-- views.py
 +-- models.py
 +-- services/
 |    +-- __init__.py
 |    +-- twitter.py
 |    +-- twitpic.py
 +-- utils.py
 +-- templates/
 +-- tests/
      +-- __init__.py
      +-- test_models.py
      +-- test_views.py
      +-- test_services.py
      +-- test_utils.py
```

○ project/__int__.py

启动应用所需的处理都写在这里，比如 WSGI 应用的入口点等。另外，models 的导入、数据库连接的初始化、设置文件的读取等都在这里进行。

○ project/views.py 以及 project/utils.py

分别实现 View 和 Utility。

○ project/models.py

实现 DomainModel 和 ApplicationModel。另外，连接 DB 所需的模块等也要事先导入到这里。

○ project/services/

在存在多个 services 时，由程序包进行整合。

○ project/templates/

存放 HTML 模板的位置。

○ project/tests/

存放单元测试（Unit Test）的位置。

另外，当项目规模非常大的时候，会由多个 project 合在一起形成一个网站。比如用户种类不同（终端用户与服务提供方的用户）或 DomainModel 完全分离时。

8.2　测试

完成设计之后，应该在动手实现前先把测试考虑好。因为如果没有一个能验证成品是否符合设计初衷的方法，我们根本无法着手去实现。反过来，只要把现成的测试摆在那里，那么我们只需以通过测试为目的去编程即可。

实现阶段的测试包括单元测试和功能单元测试。我们应该让其自动化，并且时常查看其结果。由于测试要重复执行多次，所以必须保证其足够迅速，并且可以随时随地执行。要满足“随时随地执行”这一点，就意味着这些测试不能依赖于环境。分离组件可以让不依赖于环境的测试成为可能。

本节将介绍借助 Python 中的测试工具进行测试的手法，以及从测试中提出环境依赖的技巧。

8.2.1　测试的种类

测试其实只是一个泛泛的概念，对于不同的对象或观点，其实施内容都不一样。人们常说集成测试、单元测试，但究竟是集成了什么，又是以什么为单元呢？这里我们以此思路为基础，对其进一步细分。

○ 单元测试

测试函数、方法等最小单元的测试。这个等级的测试能明确看到输入和输出，所以测试内容往往就是函数或方法的设计方案本身。该部分要利用 mock 或 dummy，把测试对象的处理单独拿出来执行，看结果是否达到预期。

○ 组件集成测试

这是集成多个函数或方法的输入输出的测试，测试时需要将多个测试对象组合在一起。由单个测试对象构成的流程已在单元测试中测试完毕，所以不参与这一步测试。对象的前后处理与单元测试一样要使用 mock 或 dummy。

○ 功能单元测试

测试用户能看得到的功能。此时用户的输入项目以及数据库等外部系统为输入的来源。输出则是向用户显示的结果、向数据库保存的内容以及对外部系统的调用。系统内部不使用 mock 和 dummy，而是全部使用正式的代码。不过，在对付某些异步调用之类的难以自动测试的部分时，需要进行一定程度的置换。外部系统方面，要准备好虚拟的 SMTP 服务器或 Web API 服务器，用以执行应用内的通信。

○ 功能集成测试

集成各功能之间输入输出的测试。这里要尽可能不去直接查看数据库内的数据，比如可以用引用类功能来显示更新类功能生成的数据。另外在与外部系统的联动方面，要借助开发专用的 API 等模拟出正式运行时的结构，然后再进行测试。这部分测试要依赖于数据库以及外部服务等诸多环境，难以自动执行，所以属于偏手动的测试。

○ 系统测试

对需求的测试。测试成品是否最终满足了所有需求。在客户验收项目时进行。

○ 非功能测试

对性能、安全等非功能方面进行的测试。借助压力测试软件进行正常 / 高峰 / 极限情况的测试，通过 XSS、CSRF 以及注入式攻击等模拟攻击来验证系统的安全性及可靠性。

本节，我们将就需开发者主要负责的单元测试、集成测试以及功能测试进行学习。

8.2.2　编写单元测试

现在我们来编写单元测试。Python 的标准库里有为编写单元测试而准备的 unittest 模块。另外，执行测试时建议使用 pytest。pytest 是一款能够自动搜索并执行测试的测试执行工具，并且

会输出详细的错误报告。

本节将为各位讲解如何用 unittest 和 pytest 生成并执行单元测试。

◉ unittest 模块

它是基于 xunit（派生自 unit、sunit 等）的测试工具及程序库。其中主要使用的是 unittest.TestCase 类。我们定义一个继承自它的类，然后在其各个方法中描述测试用例（Test Case）。测试用例要写在名称以 test 开头的方法中，具体如下。

测试用例写在名称包含 test 的方法中。

```
import unittest

class TestIt(unittest.TestCase):

    def test_it(self):
        """ 本方法是一个测试用例 """

    def test_one(self):
        """ 这是另一个测试用例 """
```

另外，TestCase 类里有用于设置环境的配置器（Fixture）和用于查看测试结果的断言方法（Assert Method）。

配置器指执行测试所需的必要前提条件的搭建以及执行完毕后的善后工作。unittest 在执行测试用例前后会分别调用各个类的 setUp 和 tearDown 方法，因此我们可以在这两个方法内描述执行测试所需的环境设置（LIST 8.1）。Python 2.7 之后的 unittest 允许使用以模块为单位的配置器。我们可以把整个模块共通的、不必每次测试都重置的环境设置流程写在模块配置器里，这能够有效削减测试的执行时间（LIST 8.2）。

LIST 8.1　各个类的配置器

```
class TestIt(unittest.TestCase):

    def setUp(self):
        """ 搭建测试环境 """

    def tearDown(self):
        """ 测试后的环境清理 """
```

setUp 在测试前被调用，tearDown 在测试后被调用。

LIST 8.2　各个模块的配置器

```
def setUpModule():
    """ 搭建测试环境 """

def tearDownModule():
    """ 测试后的环境清理 """
```

setUpModule 会在模块测试开始前被调用一次。tearDownModule 会在整个模块的测试全部结束后被调用一次。

Python 在查看测试结果时要使用 assert 语句等断言。

```
a = 1
b = 2
c = a + b
assert c == 3, "%d != %d" % (c, 3)
```

assert 语言失败时会同时显示错误信息和测试结果（失败显示 F，出错显示 E）。不过，由于 assert 语句本身只能进行 True、False 的判断，而且信息很多为定式，所以 unittest 的 TestCase 类大多拥有自己的断言方法。

最常用的断言方法当属 assertEqual。刚才那个例子改用 assertEqual 会变成下面这样。

```
def test_it(self):
    a = 1
    b = 2
    c = a + b
    self.assertEqual(c, 3)
```

◉ testfixtures 库

testfixtures 库（测试配置器）整合了多种典型配置器。里面包含生成 / 删除临时目录、更改系统日期、添加 mock/dummy 等模块，这些模块能帮助我们将单元测试与环境分离开来。

使用 testfixtures 库之前需要进行安装，安装步骤与通常的库相同（LIST 8.3）。

LIST 8.3　安装

```
$ pip install testfixtures
```

testfixtures 的 compare 函数显示出的错误信息比 unittest 的 assertEqual 还要详细。

```
from testfixtures import compare

def test_add():
    result = add(2, 3)
```

```
    compare(result, 5)
```

使用 compare 函数时，只需将比较对象用作实参直接执行即可。

另外，该比较会递归地执行到 dict 及 list 内部，并生成结果报告。

在比较下面这种复杂数据的时候，compare 函数能详细显示出 list 中的 dict 元素如何不同。

```
>>> compare([{'one': 1}, {'two': 2, 'text':'foo\nbar\nbaz'}],
...         [{'one': 1}, {'two': 2, 'text':'foo\nbob\nbaz'}])
Traceback (most recent call last):
 ...
AssertionError: sequence not as expected:

same:
[{'one': 1}]

first:
[{'text': 'foo\nbar\nbaz', 'two': 2}]

second:
[{'text': 'foo\nbob\nbaz', 'two': 2}]

While comparing [1]: dict not as expected:

same:
['two']

values differ:
'text': 'foo\nbar\nbaz' != 'foo\nbob\nbaz'

While comparing [1]['text']:
@@ -1,3 +1,3 @@
 foo
-bar
+bob
 baz
```

多行的字符串会以 unified diff 格式显示不同。可见，它会递归地根据数据类型不同选用最直观的显示方法。

另外，比较方法和结果输出可通过 testfixture.comparison.register 函数进行添加。各位想多次重复使用 assert 语句时不妨一试。

使用 Comparison 类可以一次性查看对象的属性。

```
from testfixtures import compare, Comparison as C

def test_create_object():
    result = create_object(name="dummy-object", value=4)
    compare(result, C(SomeObject, name="dummy-object", value=4))
```

ShouldRaise 可以查看发生的例外。特别是它连交给例外的传值参数都能查看到，这是 unittest 的 assertRaises 做不到的。

```
from testfixtures import ShouldRaises

def test_critical_error():
    with ShouldRaise(CriticalError('very-critical')):
        important_process(None)
```

testfixtures 提供的能应用于单元测试的实用程序还远不止这些，其他还有控制台输出、日志输出等等。加之它可以像普通的模块一样对待，所以能够在 unittest、nose、pytest 等 testrunner 上使用。

◉ 通过 pytest 执行测试

pytest 是第三方出品的测试工具。它描述测试比 unittest 要简单，而且能输出详细的错误报告。pytest 能够自动发现并执行测试，其中包括用 unittest 写的测试，所以 unittest 与 pytest 完全可以并用。

pytest 可以用 pip 安装，具体如 LIST 8.4 所示。

LIST 8.4　安装

```
$ pip install pytest
```

pytest 执行时会在指定的目录（未指定状态下则默认当前目录）下寻找测试。这个过程称为 Test Discovery。Python 程序包下的 tests 模块以及"test_**_test"等形式的名称都会被识别为测试模块。pytest 发现测试模块后会执行该模块内的测试并显示结果。

◉ 实际编写一个测试并执行

建议不要把测试放得离测试对象太远。我们将测试与测试对象放在同一个程序包内，以"test_{ 测试对象的模块名 }.py"命名该文件。接下来用 unittest 写一个测试用例（LIST 8.5）。

LIST 8.5　测试对象：bankaccount.py

```
class NotEnoughFundsException(Exception):
    """ Not Enough Funds """
```

```
class BankAccount(object):

    def __init__(self):
        self._balance = 0

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def get_balance(self):
        return self._balance

    def set_balance(self, value):
        if value < 0:
            raise NotEnoughFundsException

        self._balance = value

    balance = property(get_balance, set_balance)
```

这是一个传统的 BankAccount 教程。状态只有 _balance，balance 属性为包装。其实际的逻辑是 withdraw 和 deposit。现在我们把这些测试写出来。

LIST 8.6　测试类

```
import unittest

class TestBankAccount(unittest.TestCase):
```

这里创建一个继承了 unittest.TestCase 的测试类（LIST 8.6）。类名没有特殊要求，但最好让人能明确分辨出测试对象。比如“Test+ 测试对象类名”这种命名规则就很不错。

LIST 8.7　加载测试对象

```
class TestBankAccount(unittest.TestCase):

    def _getTarget(self):
        from bankaccount import BankAccount
        return BankAccount

    def _makeOne(self, *args, **kwargs):
        return self._getTarget()(*args, **kwargs)

    #...（略）...
```

这是用来准备测试对象的实用方法。为防止模块的副作用对其他测试产生影响，这里需要单独导入（LIST 8.8）。

LIST 8.8　测试方法

```
class TestBankAccount(unittest.TestCase):

    #...（中间省略）...

    def test_construct(self):
        target = self._makeOne()
        self.assertEqual(target._balance, 0)

    def test_deposit(self):
        target = self._makeOne()
        target.deposit(100)
        self.assertEqual(target._balance, 100)

    def test_withdraw(self):
        target = self._makeOne()
        target._balance = 100
        target.withdraw(20)
        self.assertEqual(target._balance, 80)

    def test_get_balance(self):
        target = self._makeOne()
        target._balance = 500
        self.assertEqual(target.get_balance(), 500)

    def test_set_balance(self):
        target = self._makeOne()
        target.set_balance(500)
        self.assertEqual(target._balance, 500)

    def test_set_balance_not_enough_funds(self):
        target = self._makeOne()
        from bankaccount import NotEnoughFundsException
        try:
            target.set_balance(-1)
            self.fail()
        except NotEnoughFundsException:
            pass
```

测试方法要每个方法分开描述。在 _makeOne 方法内生成测试对象的实例，备齐测试的前提条件，然后执行测试。测试结果在 assert* 方法内查看。会出现例外的测试要使用 fail 方法，

或者用 assertRaises 也行。

执行 pytest 很简单，只要在描述测试的文件（test_bankaccount.py）所在的目录下执行 py.test 即可。pytest 会自动找出并执行测试（LIST 8.9）。

LIST 8.9　执行测试

```
$ py.test
```

◉ 巧用 pytest 插件

pytest 的插件多种多样，比如收集执行数据并进行解析的插件、改变测试结果显示模式的插件，等等。除 pytest 自身包含的插件外，我们还可以选择使用第三方开发的插件，甚至自己编写插件。

○ pytest–cov

coverage 会在执行命令时收集该命令的信息。使用 pytest-cov 可以查看测试中都执行了哪些代码。虽然一味盲从这个数值是很危险的，但我们可以利用它来推断哪些部分未被测试。pytest-cov 可以通过 pip install pytest-cov 进行安装。安装完后用 --cov 选项指定要获取覆盖率的程序包。

○ xunit

它和 JUnit 一样会将测试结果保存在特定格式的文件中。在与 Jenkins 等 CI 工具联动时会用到它。它是 pytest 标配的插件，可以通过 --junit-xml 选项添加使用。

○ pdb

它会在测试发生错误时自动执行 pdb（Python 的调试器）。可以通过 --pdb 选项添加使用。

专栏　什么是覆盖率

覆盖率由百分比表示。比如测试代码执行过了程序的每一行，那么覆盖率就是 100%。这种时候，几乎不会出现新程序上线后突然无法运行的尴尬情况。不过，这一过程只是让程序跑了一遍而已，因此并不能检测出逻辑错误引起的 Bug。换句话说，覆盖率不关心代码内容究竟是什么。覆盖率是用来检查“测试代码不足、测试存在疏漏”的一个指标，“测试内容是否妥当”并不归它管。覆盖率按评测标准分为 3 个阶段（①最宽松，③最严格）。

① 指令覆盖率（简称 C0）：只要执行了所有命令即可。比如存在 if 语句，只要测试代码从 if 语句中通过即可达到 100%。

② 分支覆盖率（C1）：通过所有分支即可。如果代码中存在 if..elif..else 这样的分支，需要执行所有分支以及“不进入分支的情况”才可以达到 100%。

③ 条件覆盖率（C2）：当存在多个分支条件时，测试代码需要执行过所有情况才能达到 100%。

指令覆盖率、分支覆盖率、条件覆盖率三者之间，后者达到 100% 的难度都要高于前者。指令覆盖率比较容易达到 100%，所以我们通常希望能保证这个 100%。但是能分给编写测试代码的时间毕竟有限，如果为提升覆盖率编写大量测试，那么维护测试代码的成本将大大提升（测试代码负债化）。所以我们应该现实一点，根据眼前情况判断覆盖率应当达到多高。

另外，引入覆盖率后，大家会发现无意义的行以及意图不明确的处理都很难被覆盖，这就顺便督促了我们在编程时应尽量避免出现上述情况。覆盖率能让我们注意到一些平时注意不到的东西，所以还没接触过它的朋友请务必一试。

◉ 使用 mock

mock 是将测试对象所依存的对象替换为虚构对象的库。该虚构对象的调用允许事后查看。另外，还允许指定其在被调用时的返回值以及是否发生例外等。

mock 可以通过 pip 安装，具体如 LIST 8.10 所示。

LIST 8.10　安装

```
$ pip install mock
```

○ 虚构对象

用 mock.Mock 生成虚构对象。虚构对象的添加方法也很简单。虚构对象生成后，用任何方法都可以调用它。

```
>>> import mock
>>> m = mock.Mock()
>>> m.something("this-is-dummy-arg")
```

用 return_value 可以指定返回值。

```
>>> m.something.return_value = 10
>>> m.something("this-is-dummy-arg")
10
```

执行后可以查看到 mock 的方法的调用。

```
>>> m.something.called
True
>>> m.something.call_args
(('this-is-dummy-arg',), {})
```

此外，还能指定让其发生例外。

```
>>> m.something.side_effect = Exception('oops')
>>> m.something('this-is-dummy-arg')
Traceback (most recent call last):
 ...
Exception: oops
```

可以在测试内使用断言。

- assert_called_with
- assert_called_once_with

这些方法的作用是在测试对象的处理执行完毕后，查看其被调用的情况以及被调用时的传值参数。

```
>>> m = mock.Mock()
>>> m.something('spam') # 第一次调用
<mock.Mock object at 0x00000000025AF7F0>
>>> m.something.assert_called_with('egg')
Traceback (most recent call last):
 ...
    'Expected: %s\nCalled with: %s' % ((args, kwargs), self.call_args)
AssertionError: Expected: (('egg',), {})
Called with: (('spam',), {})
>>> m.something.assert_called_with('spam')
>>> m.something('spam') # 第二次调用
<mock.Mock object at 0x00000000025AF7F0>
>>> m.something.assert_called_once_with('spam')
Traceback (most recent call last):
  ...
   raise AssertionError(msg)
AssertionError: Expected to be called once. Called 2 times.
```

○ patch

生成虚构对象后，需要将其混入测试对象的代码之中。虚构对象用作传值参数的时候最好对付，只要将其作为传值参数交出去即可，但处理过程中要用到的类或函数就不能通过传值参数来传递了。这种时候，mock 可以在 patch 的作用范围内将模块内的类或函数临时替换为虚构对象（LIST 8.11）。

LIST 8.11　patch 装饰器的使用示例

```
@patch('myviews.SomeService')
def test_it(MockSomeService):
```

```
    mock_obj = MockSomeService.return_value
    mock_obj.something.return_value = 10
    from myviews import MyView
    result = MyView()
    assert result = 10
```

通过 patch 的传值参数指定要替换的对象。这样一来，被替换进去的 Mock 对象就会作为测试的传值参数被传进去。这个替换仅在执行被 patch 装饰器装饰的函数的过程中有效，所以不会对其他测试造成影响。

◉ 如何编写高效率的 Python 测试用例

光拿 unittest 模块来写测试用例并不能让测试变得高效。写完测试用例后要多运行几遍。另外，在修改源码时如果已经有了内容明确的相关测试用例，那么要迅速且准确地找出受影响的部分。另外，各个测试用例要分离得足够开。如果修改一个测试用例导致其他测试用例不能运行，这将会成为一种负债。

要想最大限度地巧用测试用例，需要注意以下几点。

- 尽可能简单

  要让人从测试内容中一眼看出输入和输出。

- 尽可能快，多执行几次

  单元测试多执行几遍。为实现这一点，要用pytest简化执行操作，并且保证测试本身的执行不消耗过多时间。如果执行一次就要花去10分钟，那么没人会愿意多执行几遍的。

- 分离各个测试

  保证测试数据不被多个测试用例共享。对一个测试有用的数据对其他测试不一定有用，如果测试中包含了这种没用的数据，会使输入输出变得不明确。另外，如果遇到不得不变更测试数据的情况，那么必须事先查清其影响范围。

- 不直接向模块内 import 测试对象

  如果直接向测试模块内import测试对象，那么一旦这个测试对象的import本身会带来问题，就会导致测试模块内的所有测试用例全部无法评测。此外，某些测试用例会将import失败判断为错误。还有，如果模块包含会在import时执行的代码，那么还没等测试开始，这些代码就先执行完毕了。

- 简化配置器

  在“单元测试”部分我们已经提过了，不要试图用setUp方法做完一切准备工作。setUp不是用来存放相同的处理的地方。尤其不能用setUp来生成依存于测试的数据配置器。

- 不搞通用的数据配置器

  通用的数据配置器（Data Fixture）会在测试之间建立依存性。在单元测试阶段，要保证

只在测试用例中创建必要的输入数据。

- 不过分信任虚构对象

  mock库可以轻松分离测试对象。但是不能过分依赖于mock。正因为mock简单，我们才容易被它骗。另外，检查一下是不是写了“在mock返回了mock之后返回一个mock”这种mock。这么复杂的mock真的是模仿其他组件做出来的？这种mock会让输入输出变得模糊，同时导致关联度上升。在制作mock上费脑筋是一个危险信号，此时应该重新审视设计方案。

8.2.3　从单元测试中剔除环境依赖

要保证单元测试不依赖于环境，只执行测试对象本身。否则，每当其所依赖的模块或外部系统的运作稍有变化就要对测试进行一次修改，这样单元测试眨眼间就会成为一种负债。

◉ 请求 / 响应

View 的形式有许多种。接下来我们选 Python 框架中几种常见的模式各写一个测试。

○ View 类

View 类是用类定义的 View（LIST 8.12）。它不是单纯的函数，所以可以使用属性和方法，使得结构更加多变。View 的处理在类的方法中进行定义，测试时用 dummy 替换这些方法，我们就能做到只运行测试对象的方法了。

使用 View 类时，大部分框架会用构造函数来接收请求，通过方法来进行 View 的处理。此时能很轻松地用 dummy 替换请求。另外，虽然 View 会调用服务以及读取模型，不过我们完全可以用 mock 来替换它们。在 View 的输出中，HTTP 响应是方法的返回值，它能被视为响应对象。对于这种状态，View 绝大多数时候都已经做完了 HTML 渲染。鉴于 HTML 内容的易变性，测试时最好确认一下交给模板的传值参数。

LIST 8.12　典型的 View 类

```
class MyView(object):
    def __init__(self, request):
        self.request = request

    def index(self):
        s = SomeService()
        result = s.some_method(**self.request.params)
        return render('index.html', dict(result=result))
```

为达到分离效果需要解决下面两点内容。

① 把 SomeService 替换为 mock

② 获取传给 render 的 dict 的内容

LIST 8.13　更便于测试的 View 类

```
class MyView(object):
    someservice_cls = SomeService
    def __init__(self, request):
        self.request = request

    def index(self):
        s = self.someservice_cls()
        result = s.some_method(**self.request.params)
        self.render_context = dict(result=result)
        return render('index.html', self.render_context)
```

把 SomeService 替换为 mock 的最简单方法就是把它放在类里。另外，只要把 dict 的内容放在对象里，我们就能在测试中查看它了（LIST 8.14）。

LIST 8.14　test

```
class DummyRequest(object):
    def __init__(self, params):
        self.params = params

class DummySomeService(object):
    def somemethod(self, **kwargs):
        return kwargs

class TestIt(unittest.TestCase):
    def test_it(self):
        request = DummyRequest(params={'a': 1})
        target = MyView(request)
        target.someservice_cls = DummySomeService
        result = target.index()
        self.assertEqual(target.render_context['result'], {'a': 1})
```

◉ 全局请求对象

某些框架会把请求对象当作一个线程本地化的全局对象提供给我们。更改它们需要对框架进行调整，所以必须加以注意。另外，有些时候框架根本不允许我们对请求对象进行修改。对于这类情况，框架一般都会提供一个从框架传送虚拟请求的机制，我们要利用这一机制来控制输入（LIST 8.15）。Flask 就属于这类框架。

LIST 8.15　test

```
with app.test_request_context:
    result = myview()
```

然而麻烦来了，Flask 的 view 的返回值是响应体。而且 view 是个函数，没地方存储传给模板的上下文。其实有一个地方可以用，那就是请求对象（LIST 8.16）。

LIST 8.16　myviews.py

```
def index():
    s = SomeService()
    result = s.some_method(**self.request.params)
    request.environ['render_context'] =  dict(result=result)
    return render('index.html', self.render_context)
```

把上下文添加到 request.environ 中，事后可以通过测试用例查看。此外，在框架内部使用的 ApplicationModel 等也会变得难以替换。到这里，我们自然希望 mock 等专用库伸出援手。不过别急，我们先不用库，直接混入些 dummy 试试。

```
def test_it():
    import myviews
    SomeService_orig = myview.SomeService
    try:
        myviews.SomeService = DummySomeService
        app = flask.Flask(__name__)
        with app.test_request_context:
            result = myview()

            assert flask.request.environ['render_context'] == {'a': 1}
    finally:
        myviews.SomeService = SomeService_orig
```

显然，这作为测试而言太取巧了，我们甚至需要测试一下这个测试是否能够正常运行。这种时候，使用 mock 库就简单得多。

```
@patch('myviews.SomeService')
def test_it(MockSomeService):
    myviews.SomeService = DummySomeService
    app = flask.Flask(__name__)
    with app.test_request_context:
        result = myview()

        assert flask.request.environ['render_context'] == {'a': 1}
```

由于问题的根本在于测试对象会直接返回响应体，所以我们用装饰器来避开它。

```
def render_view(name)
    def dec(view_func):
        def wraps()
            data = view_func()
            return render_template(name, data)
        return wraps
    return dec
```

但是，如果仅有上面这个处理，我们只能在测试对象加了装饰器的状态下进行访问，所以需要再作一些调整，以保证能在加了装饰器的状态下直接获取原来的函数。

```
def render_view(name)
    def dec(view_func):
        def wraps()
            data = view_func()
            return render_template(name, data)
        wraps.inner = view_func
        return wraps
    return dec
```

这样一来，我们就能在测试中直接查看返回值了。

```
@patch('myviews.SomeService')
def test_it(MockSomeService):
    myviews.SomeService = DummySomeService
    app = flask.Flask(__name__)
    with app.test_request_context:
        result = myview.inner()

        assert result == {'a': 1}
```

◉ **数据库**

SQLAlchemy 支持 sqlite 的内存数据库。使用内存数据库时，可以在不具备实际数据库的情况下测试伴随数据库连接的处理。另外，SQLAlchemy 用 DBSession 对象进行访问数据库以及取出、更新 DomainModel 的操作。

我们继续按照本章中的设计，将 View 能直接访问的 Model 限制在一个。涉及多个 DomainModel 的处理交给 ApplicationModel 来应付。这样一来，View 就只会从 DBSession 加载一个模型了。它对 DBSession 的调用是输出，从 DBSession 获取的内容是输入。

输入大致分为两种情况，即存在 DomainModel 的情况和不存在 DomainModel 的情况。在测

试对象执行前将 DomainModel 放到（或不放到）会话（session）内，这样，我们就能轻松控制这两种情况了。DomainModel 的调用方法要限制在一个。需要调用多个时交给 ApplicationModel 来处理。

由于存在实际访问数据库的行为，所以测试前后需要用配置器来设置数据库。另外，我们不必每次测试都设置一遍数据库，因此要选用模块配置器。

首先编写一个实用程序用作设置数据库的配置器（LIST 8.17）。

LIST 8.17　数据库配置器

```
def _setup_db():
    from .models import DBSession, Base
    from sqlalchemy import create_engine
    engine = create_engine("sqlite:///")
    DBSession.remove()
    DBSession.configure(bind=engine)
    Base.metadata.create_all(bind=engine)
    return DBSession

def _teardown_db():
    from .models import DBSession, Base

    DBSession.rollback()
    Base.metadata.drop_all(bind=DBSession.bind)
    DBSession.remove()
```

在实际的测试套件中，通过模块配置器调用上述实用程序（LIST 8.18）。

LIST 8.18　测试

```
def setUpModule():
    _setup_db()

def tearDownModule():
    _teardown_db()

class TestMyView(unittest.TestCase):
    def setUp(self):
        from .models import DBSession
        self.session = DBSession

    def _setup_entry(self, **kwargs):
        from .models import Entry
        entry = Entry(**kwargs)
```

```
        self.session.add(entry)
        return entry

    def test_it(self):
        e = self._setup_entry(name=u"this-is-test")

        result = self._callFUT(entry_id=e.id)

        self.assertEqual(result['name'], e.name)
```

测试数据的生成不要在 setUp 中进行。保证在测试方法内只生成该测试所需的数据。

一旦用 setUp 生成测试数据，这些测试数据就会被多个测试共享。被共享的测试数据会给各个测试之间带来依存性。

多余数据会使输入输出变得不明确。而输入输出一旦不明确，会让人很难搞清到底在测试什么。另外，当前提改变后，如果我们对测试数据进行更改，其影响的测试可能导致有问题的数据残留在程序中。所以，我们需要让单元测试只能给各个测试准备该测试所需的数据，并要在保证没有多余数据的情况下进行测试。

◉ 系统时间

我们以判断系统时间是否为月末的实用程序为例来思考一下。系统时间每次调用都会返回不同的值，所以做自动测试时要花些心思才行。这里我们暂且把系统时间放在一边，先编写一个通过传值参数接收时间并进行判断的实用程序，然后再写一个函数来给这个传值参数传递系统时间（LIST 8.19、LIST 8.20）。

LIST 8.19　实现通过传值参数接收时间并进行判断的实用程序

```
def is_last_of_month(d):
    return (d + timedelta(1)).day == 1
```

接下来进行测试。

```
def test_is_last_of_month():
    d = date(2011, 11, 30)
    assert is_last_of_month(d), "%s" % d

def test_is_last_of_month_not():
    d = date(2011, 11, 29)
    assert not is_last_of_month(d), "%s" % d
```

LIST 8.20　实现用系统时间进行判断的实用程序

```
def is_last_of_month_now():
    return is_last_of_month(datetime.now())
```

把这个也测试一遍。获取时间部分使用 testfixtures 的 Replacer 和 test_date。

```
from datetime import datetime,date

def test_is_last_of_month_now():
    with Replacer() as r:
        r.repace('util.datetime', test_date(2011, 11, 30))
        assert is_last_of_month_now()
```

这里不能替换 datetime.datetime，而是要替换掉测试对象 import 的东西（本例中测试对象使用的是 util.datetime）。在 ApplicationModel 或 DomainModel 中使用的时候，要用 mock 替换掉 is_last_of_month_now。

实用程序的功能一定要压缩到上述这个程度。

8.2.4　用 WebTest 做功能测试

确认所有组件运转正常后，就该把它们结合起来，查看功能的运作情况了。

对于 Web 应用而言，功能测试要查看从接受请求到作出响应的整个过程是否正常运作。系统内部的所有组件都直接使用正式代码，与外部系统联动的部分用 mock。mock 部分要尽可能小。另外，如果这一阶段能拿到与相当于正式运营时的示例数据，那就更应该积极测试了。

本阶段通过内存数据库、模拟请求等来控制输入输出部分。内存数据库用 SQLite 的内存数据库即可。由于 O/R 映射工具会帮我们吸收掉 RDBMS 的差异，所以用作功能测试绰绰有余。

至于模拟请求，用 WebTest 比较容易实现。

◉ WebTest

WebTest 是用于 Web 应用功能测试的库。它会对 WSGI 应用执行模拟请求并获取结果。基本上所有 WSGI 应用的测试都可以用它。

WebTest 可以用 pip 进行安装（LIST 8.21）。

LIST 8.21　安装

```
$ pip install webtest
```

LIST 8.22　使用方法

```
def test_it():
    from webtest import TestApp
    import myproject.app
    app = TestApp(myproject.app)
    res = app.get('/')
    assert "Hello" in res
```

如 LIST 8.22 所示，WebTest 中的 webtest.TestApp 可以通过构造函数的传值参数接收 WSGI 应用类型的测试对象（比如 myproject.app）。TestApp 拥有 get、post 等方法。它可以利用这些方法来执行 WSGI 应用，接收响应对象。在测试过程中，要测试响应对象的内容以及执行测试之后的数据库的状态等。

NOTE

WSGI（Web Server Gateway Interface，Web 服务器网关接口）是 PEP 3333[①] 所倡导的机制。

WSGI 将 Web 服务器与 Web 应用之间的处理定义成了简洁统一的接口，符合 WSGI 标准的服务器以及应用之间可以相互替换，对应 WSGI 的应用可以在任意对应 WSGI 的服务器上运行。

比如在 gunicorn 上运行的 Web 应用可以直接放到 Apache 的 mod.wsgi 上运行。

关于在 gunicorn 上运行 WSGI 应用的方法，我们将在第 12 章中了解。

◉ 配置器

让 WSGI 应用能通过 WebTest 调用模拟请求。另外，还要在数据库配置器和数据库中生成必要的数据以及为外部服务准备 mock。

◉ 测试用例

一个请求对应一个测试用例。设置好配置器之后，通过 WebTest 发送模拟请求。

◉ 断言

当断言的对象为 HTML 时，响应输出的内容很容易变化，所以要检查由 HTML 输出的内容是否以字符串形式包含在其中。功能的输出多为更新 DB，所以直接检查 DB 的数据即可。另外，外部系统的调用也属于输出，这部分要通过 mock 的功能来查看。

① https://www.python.org/dev/peps/pep-3333

◉ 与外部服务有关的测试

这里假设要使用外部的搜索服务。此时只将调用服务的部分替换为 mock。另外，负责生成数据的配置器要生成测试内所需的内容。

```
from mock import patch
from webtest import TestApp

def setUpModule():
    _setup_db()

def tearDownModule():
    _teardown_db()

def _init_data():
    # 在这里生成数据

def _init_search_results():
    # 在这里创建 mock 的外部服务结果

class TestWithMock(unittest.TestCase):

    def _getTarget(self):
        from app import myapp
        app = TestApp(myapp)
        return app

    @patch('othersite.search')
    def test_it(mock_search):
        """ 测试 """

        # 前提条件
        mock_search.return_value = _init_search_results()
        _init_data()

        # 准备测试对象
        app = self._getTarget()

        # 执行测试对象
        res = app.get('/?search_word=abcd')

        # 确认结果
        assert "20" in res
        mock_account.deposit.assert_called_with(q="abcd")
```

这里我们假设测试对象 myapp 在内部通过 othersite.search 函数调用了外部服务。测试时 othersite.search 被替换为了 mock。

◉ 与认证和会话相关的测试

大部分 WSGI 应用会将认证信息放在 environ['REMOTE_USER'] 里（非此类框架的原理也是相同的，只需将存储位置替换为该框架的存储位置即可）。WebTest 支持 Cookie，所以能正常执行基于 Cookie 的会话及认证的相关测试。

LIST 8.23 Cookie 的相关测试

```
from webob.dec import wsgify

@wsgify
def myapp(request):
    c = int(request.cookies.get('count', "0"))
    request.response.set_cookie('count', str(c + 1))
    return "response %d" % c

def test_it():
    from webtest import TestApp
    app = TestApp(myapp)
    res = app.get('/')
    assert res.body == "response 0"

    res = app.get('/')
    assert res.body == "response 1"
```

如果不是与认证本身直接关联，那么可以在 extra_environ 的 REMOTE_USER 里直接进行设置。下面是直接将 REMOTE_USER 传递给 extra_environ 的例子。

LIST 8.24 认证相关的测试

```
@wsgify
def myapp(request):
    if not request.remote_user:
        return HTTPFound(location="/login")

    return "OK"

def _makeOne():
    from webtest import TestApp
    return TestApp(myapp)
```

```
def _callAUT(url, params={}, method="GET", remote_user=None):
    extra_environ = {'REMOTE_USER': remote_user}
    if method == "GET":
        return _makeOne().get(
           url, params=params, extra_environ=extra_environ)
    elif method == "POST":
        return _makeOne().post(
           url, params=params, extra_environ=extra_environ)

def test_with_login():
    result = _callAUT('/', remote_user='dummy')
    assert result.body == 'OK'

def test_without_login():
    result = _callAUT('/')
    assert result.location == "/login"
```

_callAUT 方法对测试对象的调用进行了包装。传递给这个包装后的方法的 remote_user 最后将通过 WebTest 的 extra_environ 被传递给测试对象（WSGI 应用）的 environ。

8.3　通过测试改良设计

前面我们谈了一系列仅执行测试对象的测试。不知各位在写测试时是否觉得举步维艰，是否把 mock 写得非常复杂？为什么测试会变得复杂呢？要知道，组件之间的结合度越高，给测试做准备就越花时间，测试也就越复杂。光是替换 mock 都已经要绞尽脑汁，怎么可能轻松添加新功能呢？因此我们要根据测试结果来改良设计。

便于测试的设计

什么样的设计便于测试呢？首先结合度要低。模块间的关联越少，测试起来就越简单。另外就是内聚度要高。模块做的事越少测试越简单。

◉ 面向对象原则

面向对象程序设计有几个原则。遵守这些原则能让我们的设计更加便于测试。当然，有时也要勇于打破它们。不过，如果没有特殊原因，应尽量遵循面向对象原则进行设计。

○ 单一职责原则

保证一个对象只具有一项职责，一项职责只由一个对象来负责。如果只顾眼前方便而盲目

追求对象的通用性，就很容易生成大得吓人的类。所以我们应根据单一职责原则，将正确的方法放到正确的类中。名如 HogeManager 的类往往都是未经过整理的类。给类起这种名字自然无妨，但具体的处理最好别写在 HogeManager 中。具体的处理应该在 Model 中实现，然后让 HogeManager 只负责调用即可。这样做有助于提高内聚度。

○ 接口隔离原则

即不依赖于没有必要的接口。Python 的语法中没有接口的概念，但仍然会遇到类似问题。比如一旦我们进行了类检查，那么传值参数将只能接受该类或其子类。然而 Python 支持鸭子类型（Duck Typing），非子类也可以替换使用，所以最好不要做类型检验，而是根据类内是否存在某方法来判定传值参数，从而使模型结构更加灵活。

○ 开放封闭原则

对扩展开放，对修改封闭。一个模块的修改不带来额外的派生，同时模块具有可扩展性。将某个类的对象替换为其子类的对象时，不需在扩展的基础上再修改其他模块。

这类替换必须遵循特定条件，并不是说用类和继承随时都能实现。

○ 里氏替换原则

子类替换父类的过程对调用方透明。这要求我们在定义子类时，宽松对待其能接受的内容，严格把关其返回的内容。

在下述情况下，父类不可以替换为子类。

- 子类会发生父类不允许发生的例外
- 子类不接受父类已有的参数
- 子类返回父类不返回的值
- 父类中实现的内容在子类中被重载，但并未实现相关处理

如果只是想加强代码的通用性，那么应该尽量重视复用，少用继承。

○ 复用优先于继承

继承是一种紧密的结合，因为我们无法将子类与父类彻底分离。复用则不同，它也是对已有代码的一种再利用，但不会产生继承关系。另外，复用允许我们在测试时用虚拟对象替换原对象。从对象角度看，父类与子类属于同一个东西，从模块角度看则不然。

NOTE

面向对象程序设计的原则并不止这 5 个。本章仅是从活用测试结果的角度挑选了几个加以说明。

8.4　推进测试自动化

测试要做到让任何人都能随时执行。因为这种任何人都能随时执行的测试可以自动化。后面的章节中我们将介绍 CI 工具（Continuous Integration Tool，持续集成工具），它能够在我们向版本库提交代码时自动执行测试，并返回一个规整的测试结果报告。

执行测试和提交测试代码通常都是开发者的工作。此时各位不妨回想一下昨天的情况。测试用例增加了吗？测试用例中通过测试的比例变了吗？覆盖率怎么样？

自动化的 CI 工具会为我们持续保存这些测试结果。它能帮助我们更有效地利用测试结果，并且可以使用已提交至版本库的正式代码进行测试（毕竟谁都难免有忘记提交的时候），从而及早发现源码乃至开发效率上的种种问题。

CI 工具的用法我们放到后面的章节再学习。本章剩余的部分，我们用来给 CI 工具铺路，学习一下如何搭建让任何人都能随时执行测试的环境。

8.4.1　用 tox 自动生成执行测试的环境

`tox` 能便捷地为我们准备好执行测试所需的环境。`tox` 会在多个 virtualenv 环境中搭建测试环境，然后在这些环境中执行测试并显示结果。它能够把测试工具的选项及环境变量等内容统一起来，所以我们只需执行 `tox` 命令即能轻松完成所需的测试。

如下所示，可以用 pip 安装 `tox`。

```
$ pip install tox
```

安装完成后，`tox` 命令就能用了。执行 `tox` 时需要用到 tox.ini 文件，我们要在这个文件中描述测试环境的设置。各个环境的设置由所用 Python 的版本、环境变量、测试所需的库、执行测试的代码等组成。

下面是 tox.ini 文件的一个例子。

```
[tox]
envlist = py26,py27,flake8
skipsdist = true
```

```
[testenv]
deps = pytest
       webtest
       testfixtures
       mock
       -rrequirements.txt

commands = py.test

[testenv:flake8]
basepython = python2.7
deps = flake8
commands = flake8
```

[tox] 节的 envlist 用于设置测试环境列表。这里我们用到了 py27、py26 和 flake8 这 3 个环境。py27 是个特殊的名字，它会找出当前已安装的 `python2.7` 命令并生成 virtualenv。

另外，我们在 skipsdist 选项中进行了设置，保证即使没有 setup.py 也能执行测试。在没有 setup.py 的情况下，依赖库由 requirements.txt 等进行管理。关于依赖库的管理，我们将在 9.2 节再次进行学习。

[testenv] 节用来进行测试环境的设置。如果存在 [testenv:flake8] 这种指定了环境名的节，则优先采用该节的设置。环境未被特别指定时，使用 [testenv] 节的通用设置。

在这个 tox.ini 文件所在的目录下执行 `tox` 之后，以下测试将会被执行。

- 在 `python2.6` 生成的 virtualenv 内执行 `py.test`
- 在 `python2.7` 生成的 virtualenv 内执行 `py.test`
- 在 `python2.7` 生成的 virtualenv 内执行 `flake8`

另外，使用 -e 选项可以对指定环境进行测试。

```
$ tox -e py27
$ tox -e flake8
```

可见，`tox` 能将多种不同的测试执行方法统一成一个。另外，由于每个测试都被分离在各自的 virtualenv 内，所以更改一个测试的设置不会对其他测试造成影响。用 Python 2.6 和 2.7 两个版本进行测试就是很好的例子。不仅如此，即便是 Web 应用框架等大程序库的不同版本测试，`tox` 同样能发挥作用。

8.4.2　可重复使用的测试环境

要想让测试能够重复进行，必须能够随时准备出完全相同的环境。如果上一次测试生成的文件或数据残留在环境中，很可能会对新一次测试造成影响。所以测试结束后，千万记得要用 tearDown 方法清理测试环境。另外，考虑到前一次测试可能由于出错等原因以中断的形式残留在里面，最好在测试开始时和结束时都调用一遍清理方法。清理的对象包括测试用数据库、目录、外部系统的过程等。

8.5　小结

本章讲解了便于测试的设计方法以及高效的测试手法。高效的测试具备以下特点。

- 测试对象明确
- 测试要检查的内容明确
- 可任意次重复执行

进行这类测试必须剔除环境依赖。本章也一直是这样做的，让测试对象不依赖于环境，仅执行测试对象。

测试同样能验证应用的正确性，有助于品质的提升。不仅如此，便于测试的设计拥有低结合高内聚的特点，通过将各自独立的组件组合在一起来实现功能。充分独立的组件所组成的结构有着很好的可维护性和可扩展性。

兼顾到测试的设计可以为我们带来如此多的好处，何乐而不为呢？

第 9 章　Python 封装及其运用

本章将对程序包的使用方法和运用技巧进行介绍。

在第 3 章中，我们已经介绍了 Python 项目的结构。本章内容可视为其后续，为各位讲解包的使用方法，包括可用于试运行环境和正式环境的部署、为执行测试的部署等方面。只要包的使用方法和封装方法得当，测试及部署的自动化也会简单很多。另外，根据实际情况，我们往往需要满足一些常规处理之外的条件。加深对封装的理解，有助于各位应对这些特殊情况，免除为其单独花费工夫的麻烦。

本章还将进一步深入地讲解 pip 的用法。此外，我们还将学习一些其在用法上的组合技巧，帮助我们活用 pip。

9.1　使用程序包

本部分将介绍一些能应用于部署及自动测试的 pip 功能。

9.1.1　程序包的版本指定

有些时候，我们需要安装一些特定版本的程序包，比如想查看某版本下的运行情况，或者需要用到某个版本之前的最后一版。

指定程序包版本的方法有几个。以 pip install colander 为例，colander 是用于模式定义和校验的 Python 程序包。我们用几种不同方式来安装指定版本（LIST 9.1）。

LIST 9.1　指定版本的方法

```
$ pip install -U colander                 # 1.0 : 最新的稳定版
$ pip install -U colander==1.0"           # 1.0 : 指定版本
$ pip install -U "colander<1.0"           # 0.9.9 : 1.0 版以前的最后一个稳定版
$ pip install -U "colander<1.0" --pre     # 1.0b1 : 1.0 版以前的最后一个版本
$ pip install -U colander==1.0b1          # 1.0b1 : 指定版本
$ pip install -U "colander<=1.0"          # 1.0 : 1.0 版以前的最后一个稳定版（包括 1.0 版）
$ pip install -U "colander>=0.9,<0.9.9"   # 0.9.8 : 0.9 版（含）以后 0.9.9 版以前的最后一个稳定版
```

可见，指定版本的方法并不唯一。某些指定方法需要用双引号“"”括起来。这是防止不等号“<”“>”被 shell 解释为重定向。

为了便于理解，我们这里指定了 -U（--upgrade）选项，因此即便指定程序包已存在于计算

机中，系统仍会再安装一遍。

另外，pip 不会安装 alpha 版、beta 版等预发布版本的程序包。至于某版本是否为预发布版本，可以看版本号是否像 1.3a1、1.3b1 这样后面跟着 a1、b1（在 PEP 440① 中有相关定义）。要安装上述预发布版本时，需要明确指定版本号，或者附加 --pre 选项。

专栏　版本命名规范与大小关系

版本的命名规范（结构）与大小关系在 PEP 440 中有相关规定。

版本号要遵循 [N!]N(.N)*[{a|b|c}N][.postN][.devN] 的结构。N 可以为正整数或 0。(.N)* 可以重复出现任意次。被 [] 括起来的部分可以省略。a、b、c 分别代表 alpha、beta、rc。

各版本号的先后顺序自然不能根据字符串排序来定，其编号方式及排序大致如 LIST 9.2 所示。

LIST 9.2　版本的编号方式及排序

```
0.9.1
0.10
1.0a1.dev1
1.0a1
1.0b1
1.0b2
1.0c1
1.0
1.0-post1
1.0.1
```

实际上，版本号这东西我们很少会在命令行指定。大多是在 setup.py 中指定版本时才会用到它。

举个例子，假设我们为 Python 2.7 开发的 myapp 程序依赖于名叫 securelib 的外部库。然而 securelib 从 1.0 版之后才开始提供我们所需的功能，所以必须指定安装 1.0 以后的版本（LIST 9.3）。现阶段的情况是，myapp 的 setup.py 中并没有给 install_requires 指定版本号。这种情况下，如果计算机中已经装有 securelib，那么其版本将不会被更新。一旦使用者环境中安装的是不提供所需功能的旧版本，程序将无法正常运行。

LIST 9.3　在 setup.py 中指定依赖库为 1.0 以后的版本

```
setup(
    name='myapp',
```

① https://www.python.org/dev/peps/pep-0440

```
    ...
    install_requires=[
        'securelib>=1.0',
    ],
)
```

这里如果指定 securelib==1.0 又会带来别的问题。因为这样一来，只要我们使用 myapp，securelib 的版本就会被限制在 1.0。就算 securelib 因为安全问题发布了 1.0.1，myapp 的使用者也只能继续用 securelib-1.0。为避免这类问题，要尽量避免在 setup.py 中指定固定版本。

接下来了解一下另一种情况，就是指定某版本之前的版本。比如 securelib-3.0 发布，其使用的 API 发生了重大变更。此时如果安装 myapp 时安装了 3.0，那么 myapp 将无法正常工作。为防止这个情况发生，可以给 myapp 追加一条指定，指定其使用 securelib-3.0 之前的版本（LIST 9.4）。

LIST 9.4　在 setup.py 中指定依赖库为 1.0 以后、3.0 以前的版本

```
setup(
    name='myapp',
    ...
    install_requires=[
        'securelib>=1.0,<3.0',
    ],
)
```

做过上述修改后，就能让使用者安装正确版本的程序包，保证 myapp 正常运行了。

NOTE

本例中，myapp 选择使用旧版本的 securelib。然而，如果程序一直依赖于旧版本的 securelib，将无法使用今后推出的安全更新。所以根据 API 的变更重新修改、测试 myapp，让其支持 securelib-3.0 才是比较好的选择。

9.1.2　从非 PyPI 服务器安装程序包

pip 搜索程序包时默认引用 PyPI 的 URL[1]。需要引用 PyPI 以外的服务器时，有两个选项可供使用。

-i（--index-url）选项可以指定其他兼容服务器来代替 PyPI。这些服务器被称为 index 服务器，PEP 301[2]、PEP 438[3] 中对它们的规格有着严格规定。举个例子，当我们想从 PyPI 的测试服

① https://pypi.python.org/simple
② https://www.python.org/dev/peps/pep-0301
③ https://www.python.org/dev/peps/pep-0438

务器[1]安装程序包时，可以像下面这样执行。

```
$ pip install bpbook -i https://testpypi.python.org/simple
```

同样地，我们可以在公司内搭建替代 PyPI 的 index 服务器，用 pip 从该服务器上进行安装。PyPI 上公开了多种搭建 PyPI 替代服务器的方法，其中 devpi[2] 可以给每个用户设置多个索引，同时具备 PyPI 镜像功能，非常好用。

另一个用起来更方便的选项是 find-links。-f（--find-links）选项用来指定包含目标程序包链接的页面。该选项指定的页面所链接的程序包会优先于 index 服务器。如果在指定页面没有找到目标程序包，计算机则会引用 index 服务器内的资源进行安装。

```
$ pip install -f https://bitbucket.org/shimizukawa/logfilter/downloads logfilter
Downloading/unpacking logfilter
  Downloading logfilter-0.9.2.zip
  Running setup.py egg_info for package logfilter
```

像上面例子中这样，指定 -f 选项后，计算机便会检查该页面内的链接，识别程序包名称，自动获取并安装最新版本。不过，pip install 只能识别符合规定的正确包名。规定大致如下。

```
{包名}-{版本号}(-{平台})?.{扩展名}
```

指定平台的部分可以省略。如果一个程序包指定了平台（如 win32），那证明它在其他平台上是无法运行的。因此，pip 只会寻找并安装适合当前平台的程序包。

专栏　安装未存放于 PyPI 的程序包

有些程序包虽然被添加到了 PyPI，但实际的程序包文件却存放在其他地方。这种程序包无法用 pip install 程序包名的方式进行安装，所以必须指定 --allow-external 选项。

```
$ pip install --allow-external 程序包名
```

另外，如果程序包被存放在 http 环境而非 https 环境下，pip 会认为程序包有可能在通讯中被篡改，因此会中断安装。在迫不得已一定要使用这类程序包的情况下，请指定 --allow-unverified 选项。

```
$ pip install --allow-unverified 程序包名
```

---

① https://testpypi.python.org/

② https://pypi.python.org/pypi/devpi

专栏　已删除的 PyPI 镜像规格

PEP 381[①] 对 PyPI 镜像服务器规格作了规定。但随着 CDN 等技术的应用，以及 PyPI 服务器逐渐稳定，镜像服务器规格的删除在 PEP 464[②] 中被提出，并最终于 2014 年春完成删除。因此，如今已不需要用 --index-url 选项引用镜像服务器。另外，曾经的 --use-mirror 选项也于 pip-1.5 起被删除。

曾提供服务的 a.pypi.python.org、b.……之类的镜像服务器的域名早已废止。现在使用旧版本 pip，通过环境变量或设置将镜像引用设为有效后，pip 有可能不正常工作。因此请各位避免使用那些会引用镜像服务器的选项。

9.1.3　程序包的发布格式

如今上传到 PyPI 的程序包大多采用 sdist 格式。sdist 是包含程序包元数据及构建方法的存档格式。它会在每次安装时读取各环境的设置，存在 C 扩展时进行构建，然后检查所需的 Python 程序包并复制到 site-packages。另外，sdist 中其实有许多文件并不会被安装。这些工作在每个平台上实施一次就足够了。相对地，二进制包的特点是仅包含已构建完毕的 C 扩展和 Python 程序包，仅解压程序包的存档即可完成安装。

Python 的二进制包长期以来使用着 setuptools 实现的 egg 格式。但是 pip 并不支持 egg 格式的程序包，这导致该格式在普及的道路上一直停滞不前。随着 Python 封装的规范化，PEP 427[③] 提出了一个能克服 egg 缺点的 wheel 格式。加之 pip 也开始支持 wheel 格式，所以其很快在部署等方面得到应用。

已上传到 PyPI 的 wheel 格式程序包可以通过 pip 直接安装。接下来，我们以安装 Django 为例来学习一下。

```
$ pip install django
Downloading/unpacking django
  Downloading Django-1.7.1-py2.py3-none-any.whl (7.4MB): 7.4MB downloaded
  Storing download in cache at /var/pip-chache/https%3A%2F%2Fpypi.python.
org%2Fpackages%2Fpy2.py3%2FD%2FDjango%2FDjango-1.7.1-py2.py3-none-any.whl
Installing collected packages: django
Successfully installed django
Cleaning up...
```

① https://www.python.org/dev/peps/pep-0381

② https://www.python.org/dev/peps/pep-0464

③ https://www.python.org/dev/peps/pep-0427

可以看到计算机从 PyPI 下载了 wheel 格式的程序包。pip 是从 1.5 版以后开始正式支持 wheel 格式的。这里我们将 pip 的版本降回 1.4，然后再安装 Django 试试。

```
$ pip install django
Downloading/unpacking django
  Downloading Django-1.7.1.tar.gz (7.5MB): 7.5MB downloaded
  Storing download in cache at /home/aodag/.pip/eggs/https%3A%2F%2Fpypi.python.
org%2Fpackages%2Fsource%2FD%2FDjango%2FDjango-1.7.1.tar.gz
  Running setup.py egg_info for package django

    warning: no previously-included files matching '__pycache__' found under
directory '*'
    warning: no previously-included files matching '*.py[co]' found under dire
ctory '*'
Installing collected packages: django
  Running setup.py install for django

    warning: no previously-included files matching '__pycache__' found under
directory '*'
    warning: no previously-included files matching '*.py[co]' found under
directory '*'
    changing mode of build/scripts-2.7/django-admin.py from 644 to 755
    changing mode of /home/aodag/works/bpbook2015/djangoenv/bin/django-admin.
py to 755
    Installing django-admin script to /home/aodag/works/bpbook2015/djangoenv/bin
Successfully installed django
Cleaning up...
```

可以看到，这种情况下的安装是通过 tar.gz 存档的 sdist 进行的。另外，相较于通过 wheel 进行安装，通过 sdist 进行安装时，下载后的处理要多出很多。

对于 Django 这种不包含 C 扩展的程序包而言，上面的差距还算可以接受。可是一旦换成 Pillow 这种包含 C 扩展，且 C 扩展需要大量依赖库的程序包时，差距就无法忽视了。现在 PyPI 上提供了支持各种 Python 版本及 CPU 的 wheel 格式 Windows 版 Pillow 程序包，即便是在难以准备编译器的 Windows 环境下，也只需要通过 pip 进行安装即可开始使用 Pillow 了。

专栏　wheel 程序包的文件名

PEP 427[①] 中规定了 wheel 程序包的命名规则。自此，wheel 程序包依赖于哪个版本的 Python 以及哪个 OS，仅凭包名即可一目了然。

① https://www.python.org/dev/peps/pep-0427

其命名与解释规则如下。

```
{distribution}-{version}(-{build tag})?-{python tag}-{abi tag}-{platform tag}.whl
```

LIST 9.5 是一些已上传到 PyPI 的程序包的文件名。

LIST 9.5　wheel 程序包文件名示例

```
bpmappers-0.7-py2-none-any.whl
Django-1.7.1-py2.py3-none-any.whl
MarkupSafe-0.23-cp27-none-linux_x86_64.whl
Pillow-2.6.1-cp32-cp32m-macosx_10_6_intel.macosx_10_9_intel.macosx_10_9_x86_64.
whl
```

9.1.4　生成 wheelhouse 的方法

wheel 格式的程序包方便好用，但我们前面也说了，PyPI 上大部分程序包都是 sdist 格式。为方便今后在 CI 工具上和部署时能有 wheel 可用，我们来学习一下如何根据 sdist 生成 wheel。pip 可以通过 wheel 命令将 PyPI 上只提供了 sdist 的程序包转换成 wheel 格式并保存在本地计算机中。本地的 wheel 格式程序包默认保存在 wheelhouse 目录下，也正因为如此，我们习惯将所有保存 wheel 格式程序包的目录都称为 wheelhouse（与实际目录名称无关）。使用 wheel 命令前，需要先安装 wheel 包。

```
$ pip install wheel
```

接下来，我们以生成 Pillow 的 wheel 为例实际生成一个 wheelhouse。

```
$ pip wheel pillow
Downloading/unpacking pillow
  Downloading Pillow-2.6.1.tar.gz (7.3MB): 7.3MB downloaded
  Running setup.py egg_info for package pillow

 ...

Building wheels for collected packages: pillow
  Running setup.py bdist_wheel for pillow
  Destination directory: /home/aodag/works/bpbook2015/wheelhouse
Successfully built pillow
Cleaning up...
```

```
$ ls wheelhouse/ -l
-rw-r--r-- 1 bp bp 714085 11月 10 19:31 Pillow-2.6.1-cp27-none-linux_x86_64.whl
```

可见，pip 和 wheel 组合使用之后，可以将只提供了 sdist 的程序包转换为 wheel 格式保存。

> **NOTE**
>
> 生成并保存在 wheelhouse 里的 wheel 依赖于执行 wheel 命令的平台。库中包含 C 扩展的 wheel 不具备跨平台通用性。比如正式服务器使用 Linux 但开发者使用 OS X，就需要准备一个与正式服务器同平台的 CI 服务器，在上面执行生成 wheelhouse 的任务。

9.1.5　从 wheelhouse 安装

现在我们来学习如何安装 wheelhouse 内的程序包。最简单的方法就是用 pip install 直接指定 wheelhouse 目录下的文件。

```
$ pip install wheelhouse/Pillow-2.6.1-cp27-none-linux_x86_64.whl
Unpacking ./wheelhouse/Pillow-2.6.1-cp27-none-linux_x86_64.whl
Installing collected packages: Pillow
Successfully installed Pillow
Cleaning up...
```

由于 C 扩展的编译等处理在这个阶段早已完成，所以安装会十分迅速。不过，wheel 格式程序包的文件名通常比较复杂，每次安装都手动输入实在麻烦。

其实，只要用 -f(--find-links) 指定 wheelhouse 目录，程序包名部分就可以只写 Pillow 了。

```
$ pip install -f wheelhouse pillow
Downloading/unpacking pillow
Installing collected packages: pillow
Successfully installed pillow
Cleaning up...
```

保险起见，我们添上 --no-index 选项以防 wheelhouse 以外的目录被引用。

```
$ pip install --no-index -f wheelhouse pillow
Ignoring indexes: https://pypi.python.org/simple/
Downloading/unpacking pillow
Installing collected packages: pillow
Successfully installed pillow
Cleaning up...
```

事先在 wheelhouse 目录下生成的 wheel 格式程序包都可以通过上述步骤进行安装。只要将所有依赖库全都放到 wheelhouse 下，我们就可以脱机完成环境搭建。另外，由于安装时所需的处理极少，所以非常适合需要多次重复搭建相同环境的情况。为部署和 CI 工具搭建环境时，请

务必记得运用 wheelhouse 目录。

9.2　巧用程序包

在需要多次重复安装的时候，wheel 格式的程序包显得十分便捷。接下来我们学习一下如何将其巧用到 CI 和部署中去。

9.2.1　私密发布

有时候我们需要用到未在 PyPI 上公开的程序包，比如不能在 PyPI 上公开的公司内部库，或者一些经过修改但尚未正式发布的库等。

这种时候，可以借助版本库服务器提供的功能做私密发布。私密发布一般用来做程序包公开到 PyPI 前的测试，以及给一些不想添加到 PyPI 但仍需安排版本号的程序包做内部公开。前面我们学习了直接从版本库进行安装的方法，使用者可以通过这个方法获取目标版本的程序包，其过程与程序包作者本人的意图基本无关。

Github 和 Bitbucket 会以标签名或分支名为单位提供 zip 或其他格式的源码文件。同时，通过使用 pip 指定包含程序包名称的完整 URL，可以绕过索引服务器直接安装程序包。将这两个组合在一起，我们就能安装已私密发布但尚未在 PyPI 发布的程序包了。

举个例子，有个名为 logfilter 的工具，其源码在 Bitbucket 中公开管理。这个工具并未在 PyPI 上公开程序包，因此不能通过 pip install logfilter 进行安装。但是，如果我们像 LIST 9.6 这样直接指定 URL，就能够完成安装了。

LIST 9.6　安装已私密发布的程序包

```
$ pip install https://bitbucket.org/shimizukawa/logfilter/get/logfilter-0.9.2.zip
```

9.2.2　巧用 requirements.txt

用 `pip freeze` 命令可以查看通过 pip 安装的依赖库的内容。如果将这些内容保存在文件中，再用 `pip install` 的 -r 选项指定这个文件，在该环境中安装的库就可以在其他环境中被还原。

```
$ pip freeze > requirements.txt
$ pip install -r requirements.txt
```

多数项目都将依赖库的列表保存在了 requirements.txt 文件中。不过，requirements.txt 能做

的事还远不止于此。

requirements.txt 内部可以引用其他 requirements.txt。比方说我们要把只在开发时才需要的库和执行时需要的库分开管理。此时，我们需要把测试运行器和配置器工具写在 tests-require.txt 文件里，把执行时需要的库写在 requirements.txt 文件里。这种情况下，只要准备一个如 LIST 9.7 所示的 dev-requires.txt，就能通过一条 pip install -r dev-requires.txt 完成开发者环境的准备工作。

LIST 9.7　dev-requires.txt

```
-r requirements.txt
-r tests-require.txt
```

另外，建议将 --allow-external 和 --allow-unverified 等常用选项也一并写在 dev-requires.txt 中。这样可以将选项与 requirements.txt 分离，免得在使用 pip freeze > requirements.txt 自动生成文件时丢失选项。

9.2.3　requirements.txt 层级化

接下来，我们考虑对 requirements.txt 进行分割。首先，requirements.txt 是许多工具默认识别的文件名，所以这个文件名的重要性最高，要用来保存执行时所需程序库的列表。接下来考虑开发者使用的工具。这部分可以按用途分为测试工具、文档生成、模式迁移等多个类别。我们为每个用途分别准备一个文件。

LIST 9.8 ～ LIST 9.12 表示的就是一个文件分割的例子。

LIST 9.8　requirements.txt

```
pyramid==1.5.2
sqlalchemy==0.9.8
psycopg2==2.5.4
```

LIST 9.9　tests-require.txt

```
pytest==2.6.4
pytest-cov==1.8.1
coverage==3.7.1
webtest==2.0.6
```

LIST 9.10　docs-require.txt

```
Sphinx==1.3b2
```

LIST 9.11　db-requires.txt

```
alembic==0.6.7
psycopg2==2.5.4
```

LIST 9.12　dev-requires.txt

```
--no-index
-f http://devpi/+myproject/simple
-r requirements.txt
-r tests-require.txt
-r docs-require.txt
-r db-requires.txt
```

这样设置下来之后，就不会在执行时安装一些没用的库了。同时，tests-require.txt 等还可以拿到其他项目之中重复利用。

9.2.4　为部署和 CI+tox 准备的 requiremests

前面我们学习过，用 pip 能够将依赖库以 wheel 格式保存在 wheelhouse 里。为方便将它们运用到 CI 和部署当中，我们要简化这些依赖库的安装流程。

用 pip 管理程序包的情况下，依赖库会记录在 requirements.txt 中。这里我们跟前面一样，在 requirements.txt 里只保存依赖库的列表。

然后，为了在搭建环境时只从 wheelhouse 获取这些库，我们像 LIST 9.13 这样编写 dev-requires.txt，将库的获取位置限定为 wheelhouse。

LIST 9.13　dev-requires.txt

```
-f wheelhoouse
--no-index
-r requirements.txt
```

有了这些文件之后，就可以用下述方法从 wheelhouse 进行安装了。

```
$ pip install -r dev-requires.txt
```

接下来，我们把它应用到我们在 8.4.1 节学习过的测试环境工具 tox 上。tox 可以在 virtualenv 中为每个测试环境搭建一个虚拟环境，如果能通过 wheel 格式给各个环境安装依赖库，那么必将提升搭建环境的效率。

tox 可以在 tox.ini 文件中指定安装到环境的库（LIST 9.14）。由于这部分是直接交给 pip 处理的，所以指定的东西不一定非要是库，还可以是 requirements.txt 等文件。

LIST 9.14　tox.ini

```
[testenv]
deps = -rdev-requires.txt
```

这样设置下来之后，计算机就会根据 dev-requires.txt 从本地的 wheelhouse 安装依赖库。这里有一点需要注意，tox 不会查看 dev-requires.txt、requirements.txt 这类文件的内容，所以如果我们对这些文件进行了编辑，需要手动执行带 --recreate 选项的 tox 命令。

```
$ tox --recreate
```

虽然难以避免上述这类麻烦，但对于某些需要频繁重置环境的 CI 工具而言，它仍是一个非常省时省力的方法。特别是使用 travis.ci 等对执行时间有限制的工具时，缩短搭建环境的时间显得十分重要。

9.2.5　通过 requirements.txt 指定库的版本

我们在用 `pip freeze` 生成的 requirements.txt 文件中指定了库的版本。通过 requirements.txt 指定版本的方法有很多种，这里我们学习直接指定版本的方法。对于程序库而言，即便只是修复 Bug，也可能导致其运行出现变化。因此，开发、测试、正式环境要保证使用同版本的库。CI 工具可以用 tox 的 --recreate 选项来严格按照 requirements.txt 中描述的库重新搭建环境。另外，想知道环境中是否安装了未包含在 requirements.txt 中的库时，可以通过 `pip freeze` 指定 requirements.txt 进行查看（LIST 9.15）。

LIST 9.15　查看是否安装了未包含在 requirements.txt 中的库

```
$ pip freeze -r requirements.txt
```

随着程序库版本不断更新，wheelhouse 内的文件会越积越多。我们可以通过下述方法将未包含在 requirements.txt 中的库转移到其他目录。

```
$ mv wheelhouse wheelhouse.old
$ pip wheel -r requirements.txt -f wheelhouse.old
```

对正式服务器而言，直接从当前活动环境中撤走程序库会带来很高的风险。这时就需要花些心思来规避风险了，比如在 virtualenv 上重新搭建一个环境，然后切换 nginx 等反向代理的连接。

9.3 小结

本章介绍了程序包的使用方法和使用技巧。正如本章所说的，巧用 pip 可以为测试和部署的自动化减轻很多负担。希望各位能以本章介绍的方法为参考，研究出适合各自项目的结构。

第 10 章　用 Jenkins 持续集成

不知各位有没有遇到过这种事——在自己的环境里运行一切正常的程序放到开发环境和正式环境中却突然无法运行了。如果有，那么在导致无法运行的原因中，有没有遗漏提交或者忘记安装关键程序呢？

这些恼人的低级错误往往会导致我们被经理叫去面谈，出错经过被刨根问底，最后被迫"写个应对方案出来"，心情沉重好几天。麻烦的是，就算我们在发现问题的头几天能提前注意，暂时避免问题的重复发生，但是随着时间推移早晚还会再犯，结果还是得对着经理铁青的脸咽苦水。相信不少人都有过类似经历吧？

如果每次修改程序都重新构建、测试、检查测试结果，我们或许能避免上述失误。但是，人类是一种不擅长单调重复劳作的生物。所以，我们需要借助工具让这一系列工作的执行自动化、定期化。

这种自动化、定期化执行的解决方案称为持续集成（Continuous Integration，CI）。这个解决方案极大降低了各种人为失误（粗心大意、重复劳作的惰性等）带来的风险。本章将以 Django 框架的 Web 应用为例，讲解如何用 CI 工具 Jenkins 实现持续集成。Django 的相关知识请参考第 14 章。

10.1　什么是持续集成

10.1.1　持续集成的简介

◉ 开发流程中存在的风险

持续集成是一种让计算机自动地任意次重复整个开发流程（编译、测试、汇报等）的开发手法，一般简称为 CI。由于其频繁重复整个开发流程，所以能帮助开发者提早发现问题。

为方便理解持续集成，现在我们把从写代码到向执行环境发布的整个开发流程大致分为以下 3 个阶段。

① 编写源码。修改已有代码
② 提交、push
③ 进行发布

在②和③的过程中容易出现下述问题，必须严加注意。

- 遗漏提交。忘记提交要发布的程序
- 遗漏合并。push 到 default 的时候忘记合并，出现多头现象
- 遗漏安装。编写的程序需要特定 Python 模块才能运行，但是忘记安装该模块了

对于这类问题，只要事先准备好测试代码，在程序发布到目标环境之前执行测试，基本都能被发现。

◉ 长期开发中容易出现的问题

我们都希望测试代码在发布之后仍能够长期继续使用。毕竟开发是一个持续的过程，一次发布之后还会有下一次发布，并不是发布一次就结束了。

在开发长期持续的过程中，我们会遇到下面这些问题。

○ 忘记以前发布的代码

肩负程序修改任务的开发者对源码的修改十分敏感，会频繁地执行测试代码。可是，一旦修改任务结束，问题关闭，就很少有人再愿意回过头去手动执行那些已经通过的测试了。

○ 只关心自己负责的部分

大部分时候，我们会去执行与自己负责部分相关的测试代码，但不会关心责任范围以外的模型的测试。然而，我们对程序的修改有时会“枪毙”掉某些看似毫无关系的模块的测试结果。估计各位之中有不少人有过类似经历。

可见，我们很难通过人为的注意和操作来确认整个开发流程，而且这样做也是一种时间和劳动力的浪费。如果项目代码量很大，那么执行所有测试代码必然消耗大量时间，如果是在自己的环境中执行，这根本不现实。所以这类工作最好交给计算机去做。

◉ 一天内多构建几次能很快发现问题

一天内多次定期 pull 源码并运行测试代码可以帮助我们及早发现问题。与其等到后期一次性找出一大堆问题，不如在产生问题时及早处理来得方便和放心。这一做法被我们称为持续集成。

> **NOTE**
>
> 持续集成是 XP（Extreme Programming，极限编程）的最佳实践之一。原文出自 Martin Fowler[①] 的论文，各位可通过下述 URL 查看[②]。

---

① 国际著名的面向对象分析设计、UML、模式等方面的专家，敏捷开发方法的创始人之一，现为 ThoughtWorks 公司的首席科学家。——编者注

② 中文版论文请参考如下网址：http://www.cnblogs.com/cloudteng/archive/2012/02/25/2367565.html。——编者注

> Continuous Integration
> http://www.martinfowler.com/articles/continuousIntegration.html

10.1.2　Jenkins 简介

Jenkins 是基于 Java 的开源 CI 工具，其安装和操作都很简单。另外，Jenkins 不仅能在面板上轻松看出 Job 成功或失败，还可以借助通知功能将结果以邮件或 RSS 订阅的形式发给用户。

除此之外，Jenkins 还允许通过插件进行功能扩展，所需功能可以随用随添加，而且还支持主从式集群，能够轻松地横向扩展。

下述流程是用 Jenkins 实现持续集成的一个例子。本章我们将根据下述流程进行学习。

- 为 Job 的执行制定日程表
- 签出源码
- 通过 shell 脚本执行测试（包括检测覆盖率）并输出结果
- 通过 shell 脚本构建文档
- 统计测试结果和覆盖率
- 统计 TODO
- 发送邮件通知 Job 的结果
- 将 Job 的结果保存至 Slack

10.2　Jenkins 的安装

我们先从 Jenkins 的安装开始学习。Jenkins 发布了面向 Windows 和 OS X 的安装包。此外，还可以通过包管理器安装 Ubuntu/Debian、Red Hat/Fedora/CentOS、FreeBSD 等。我们可以根据构建对象的环境选择相应的版本。本书将以 Ubuntu 14.04 上的安装与运行为基准向各位讲解 Jenkins。

10.2.1　安装 Jenkins 主体程序

现在来安装 Jenkins 的主体程序。首先执行下述命令添加用于 apt 的公共密钥。

```
$ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
```

然后打开 apt 的设置文件 /etc/apt/sources.list，将下述语句添加到最后一行。

```
deb http://pkg.jenkins-ci.org/debian binary/
```

最后执行下述命令，Jenkins 就会被安装到计算机中。

```
$ sudo apt-get update
$ sudo apt-get install jenkins
```

安装完成之后系统会自动生成 jenkins 用户，Jenkins 也会启动。比如启动 Jenkins 的服务器 IP 地址为 10.0.0.1，那么只要访问 `http://10.0.0.1:8080` 即可看到 Jenkins 的首页。/var/lib/jenkins 是 Jenkins 用户的主目录，该目录下保存着设置文件和工作目录。

NOTE

Gunicorn、Tomcat 等运行着 Web 容器的环境可能已经占用了 8080 端口，这种时候，Jenkins 无法在默认设置状态下提供服务。要想使用 Jenkins，必须更改 Jenkins 启动时的 HTTP 端口号，比如用文本编辑器打开 /etc/default/jenkins，将 HTTP_PORT 的值从默认的 8080 改成其他值。

10.2.2　本章将用到的 Jenkins 插件

Jenkins 可以通过插件进行功能扩展，因此广大用户为适应各种用途开发了大量插件。本章我们将用到下述 4 个插件。

① Mercurial Plugin：使用源码管理系统（SCM）Mercurial 时所需的插件
② Cobertura Plugin：生成源码覆盖率报告时所需的插件
③ Task Scanner Plugin：统计源码中的 TODO 时所需的插件
④ Slack Plugin：用来向 Slack 发送通知的插件

插件的安装可以通过“系统管理”界面的“管理插件”进行。这里为方便说明，我们来了解一下通过 CLI（Command Line Interface，命令行接口）进行安装的方法。

首先下载 Jenkins 的 CLI 工具。

```
$ wget http://localhost:8080/jnlpJars/jenkins-cli.jar
```

通过 CLI 工具安装插件。

```
$ java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin tasks
mercurial slack cobertura
```

重启 Jenkins，使刚安装的插件生效。

```
$ sudo service jenkins restart
```

方便好用的 Jenkins 插件很多，除了这里介绍的几种以外，还有 JobConfigHistory Plugin、Email-ext Plugin、Timestamper Plugin，等等。

下面是这些插件的官方维基百科。

JobConfigHistory
　https://wiki.jenkins-ci.org/display/JENKINS/JobConfigHistory+Plugin
Email-ext Plugin
　https://wiki.jenkins-ci.org/display/JENKINS/Email-ext+plugin
Timestamper Plugin
　https://wiki.jenkins-ci.org/display/JENKINS/Timestamper

10.3　执行测试代码

现在我们拿一个简单的 Python 项目的测试代码作为例子执行一下。Python 项目由版本控制系统保存，Jenkins 将从版本控制系统获取 Python 项目并执行测试。

10.3.1　让 Jenkins 运行简单的测试代码

首先，我们写一个包含 unittest 模块的简单的测试代码，让 Jenkins 来运行它（LIST 10.1、LIST 10.2）。

LIST 10.1　foo.py

```
# -*- coding:utf-8 -*-

def divide(num1, num2):
    """ 对传值参数做除法的简单函数
    """
    return num1 / num2
```

LIST 10.2　test_foo.py

```
# -*- coding:utf-8 -*-
import unittest

import foo

class SimpleTest(unittest.TestCase):
    """ 测试做除法的函数
```

```
    """
    def test1(self):
        self.assertEqual(foo.divide(2, 2), 1)

    def test2(self):
        self.assertEqual(foo.divide(0, 1), 1)

if __name__ == "__main__":
    unittest.main()
```

将这两个文件保存在 Mercurial 的版本库里，放在 Jenkins 服务器上的 /var/hg/simple_test 目录下。

10.3.2　添加 Job

接下来给 Jenkins 添加 Job。Jenkins 的 Job 可以指定多种类型的任务，比如构建项目、签出代码等。现在请打开 Jenkins 面板侧边菜单的“新建”界面，如图 10.1。

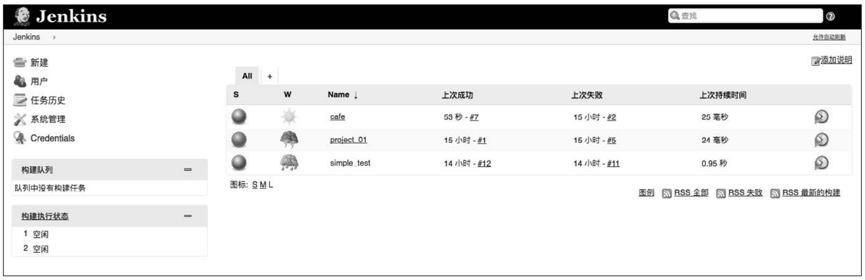


图 10.1　Jenkins 的面板

在“新建”界面的“Item 名称”处输入 Job 名，选择 Job 种类。如图 10.2，这里我们输入 simple_test 作为 Job 名，种类则选择“构建一个自由风格的软件项目”，然后点击界面下部的“OK”按钮。

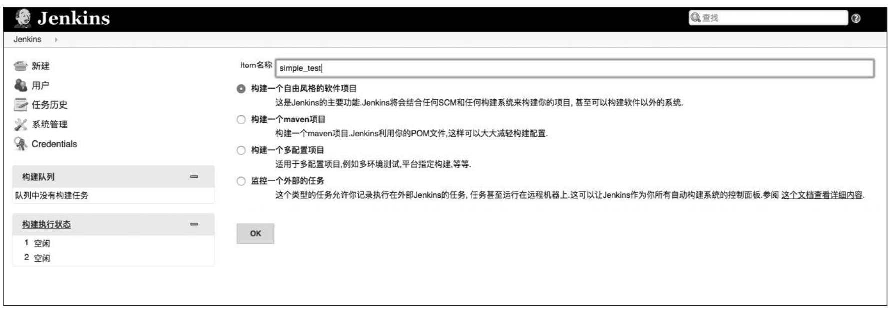


图 10.2　新建界面

◉ **指定源码管理系统**

在 Job 设置界面的“源码管理”一栏中选择 Mercurial，随后会显示 3 个输入框，我们作如下输入。

① Repository URL：输入版本库的 URL
　输入 /var/hg/simple_test
② Branch：输入 Mercurial 的分支名。留空则使用 default
　这次我们要用的是 default，因此留空
③ 源码库浏览器：指定用于查看版本库更改的工具
　现阶段先用默认的“自动”

NOTE

关于版本库 pull 出错

Jenkins 的 Job 从 Mercurial 上签出源码时，经常由于忘记设置版本库权限而导致 pull 出错。

安装 Jenkins 时会自动创建 jenkins 用户，执行 Job 的都是这个用户，所以一定要记得给这个用户权限。

另外，有时即便已经有权限，在执行 hg pull 时仍会遇到“不可信的用户”“不可信的群组”这类错误。这种情况一般出现在版本库的“.hg/hgrc”所有者与执行 hg 命令的用户不一致的时候。只要在 jenkins 用户的 $HOME/.hgrc 的 trusted 节中进行设置即可解决该问题。比如“不可信的用户”为 foo、“不可信的群组”为 bar 时，则需在 hgrc 作以下设置。

```
[trusted]
user = foo
group = bar
```

此外，trusted 节的 user、group 元素可以一次指定多个值，相邻两值之间用逗号隔开。

◉ **制定日程表**

然后我们来设置另一个重要项目——构建触发器。这个项目能像 cron 一样制定日程表，比如让 Job 定期执行，或者在其他 Job 结束后执行等。这里我们选择“Build periodically”。勾选之后会出现“日程表”输入框，我们在这里设置 Job 的执行间隔，具体方法与设置 crontab 时一样。比如我们想每个小时的 0 分时执行 Job，则写成下面这样。

```
0 * * * *
```

此外，在 Jenkins 的 Job 设置界面里，每一项后面都有为用户准备的帮助。各位如果遇到看不懂的项目，可以点击相应的帮助图标作参考。

◉ **设置执行测试代码的命令**

最后，在“构建”选项卡处设置执行测试代码的命令。点击“增加构建步骤”之后即可选择“Execute shell”。然后在文本框中描述下述命令，如图 10.3 所示。

```
python test_foo.py
```

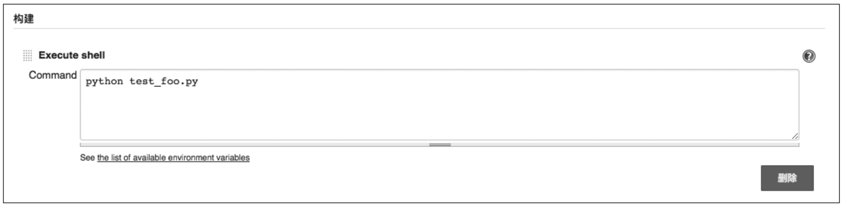


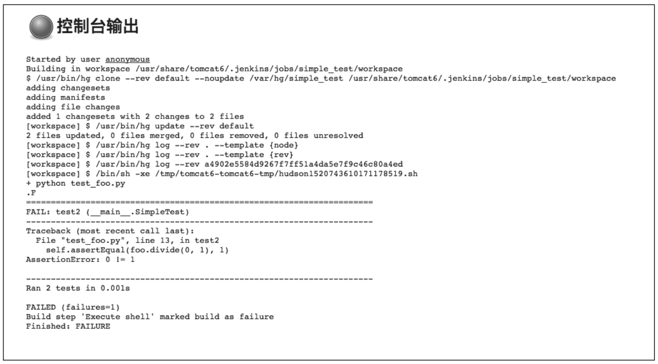
图 10.3　构建步骤“Execute shell”

只要把我们写 unittest 的测试用例时手动执行的命令写在这里即可。完成上述所有步骤之后点击“保存”按钮进行保存，接下来 Jenkins 将在我们指定的时刻执行 Job。

10.3.3　Job 的成功与失败

Jenkins 的 Job 可以从管理界面手动启动。现在我们试着点击面板上的“立刻构建”，启动刚才创建的 Job。

随后各位看到的应该是一个红色图标，表示“失败”。我们可以点击左侧栏“控制台输出”链接查看执行结果（图 10.4）。Jenkins 会以控制台日志的形式记录每次执行 Job 的过程，便于用户调查失败原因。



控制台输出

```
Started by user anonymous
Building in workspace /usr/share/tomcat6/.jenkins/jobs/simple_test/workspace
$ /usr/bin/hg clone --rev default --noupdate /var/hg/simple_test /usr/share/tomcat6/.jenkins/jobs/simple_test/workspace
adding changesets
adding manifests
adding file changes
added 1 changesets with 2 changes to 2 files
[workspace] $ /usr/bin/hg update --rev default
2 files updated, 0 files merged, 0 files removed, 0 files unresolved
[workspace] $ /usr/bin/hg log --rev . --template {node}
[workspace] $ /usr/bin/hg log --rev . --template {rev}
[workspace] $ /usr/bin/hg log --rev a4902e5584d9267f7ff51a4da5e7f9c46c80a4ed
[workspace] $ /bin/sh -xe /tmp/tomcat6-tomcat6-tmp/hudson1520743610171178519.sh
+ python test_foo.py
.F
======================================================================
FAIL: test2 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_foo.py", line 13, in test2
    self.assertEqual(foo.divide(0, 1), 1)
AssertionError: 0 != 1

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

图 10.4　通过“控制台输出”查看 Job 执行失败的内容

从日志中一眼就能看出测试结果为 FAILURE。失败的原因是测试代码有问题，所以接下来，我们要将测试用例修改成 LIST 10.3 这样，然后重新启动 Job。

LIST 10.3　test_foo.py

```
# -*- coding:utf-8 -*-
import unittest

import foo

class SimpleTest(unittest.TestCase):
    """ 测试做除法的函数
    """
    def test1(self):
        self.assertEqual(foo.divide(2, 2), 1)

    def test2(self):
        self.assertEqual(foo.divide(0, 1), 0)

if __name__ == "__main__":
    unittest.main()
```



图 10.5　执行成功时的“控制台输出”

如图 10.5，这次看到的应该是蓝色图标，测试结果为 SUCCESS。

10.4 测试结果输出到报告

经过前面的安装与设置，我们已经可以用 Jenkins 创建 Job 并执行测试代码了。不过，实际开发中的测试用例要比这个多得多，而且单纯从面板上查看测试成功 / 失败并不能满足开发的需求。

Jenkins 可以读取指定格式（xUnit 格式）的 xml 文件并将测试结果以报告形式显示在屏幕上。使用 Python 的 pytest 不仅能够一次性执行多个测试用例，还能将结果以 xUnit 格式输出。所以这里我们借助 pytest 来输出测试结果的报告。

10.4.1 安装 pytest

可以用 pip 安装 Pytest。

```
$ pip install pytest
```

10.4.2 调用 pytest 命令

安装完 pytest 后就可以调用 `py.test` 命令了。我们先移动到放有 test_foo.py 的目录下，然后执行如下命令。

```
$ py.test --junit-xml=test_result.xml
```

该命令会输出一个 xUnit 格式的 test_result.xml 文件。现在只要让 Jenkins 读取它，我们就能从屏幕中看到测试结果的报告了。

10.4.3 根据 pytest 更改 Jenkins 的设置

首先给 Jenkins 设置 virtualenv，安装 pytest。

```
$ sudo su - jenkins
$ virtualenv .virtualenv/simple_test
$ source .virtualenv/simple_test/bin/activate
$ pip install pytest
```

然后来更改 Jenkins 的设置。需要修改的只有“Execute shell”中指定的命令行操作内容。原本我们是用 python 命令直接执行 unittest 的测试用例的，这里替换成调用上面提到的 `py.test` 命令，如图 10.6。



图 10.6　根据 pytest 修改命令

接下来设置“构建后操作”选项卡，让 Jenkins 读取 `py.test` 命令输出的 xml 文件，如图 10.7。

点击“增加构建后操作步骤”按钮，选择“Publish JUnit test result report”。随后该选项卡内会自动生成一个“Publish JUnit test result report”表格项，我们在“测试报告（XML）”里指定 test_result.xml。

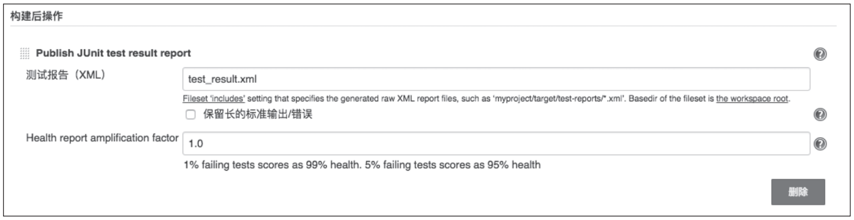


图 10.7　设置读取测试报告

现在再点击“立刻构建”，则将显示如图 10.8 的报告。

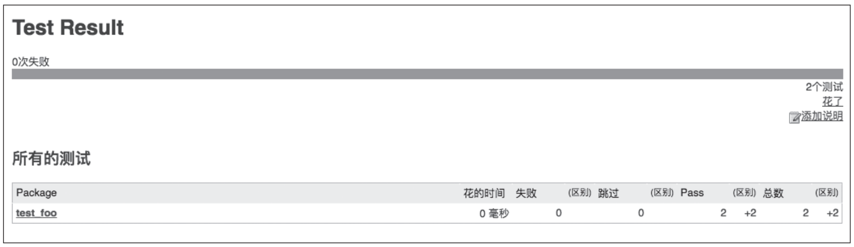


图 10.8　测试结果报告

10.5　显示覆盖率报告

现在我们已经能通过 Jenkins 界面查看测试用例执行结果的报告了。接下来我们看看如何用 pytest-cov 获取代码的覆盖率。

10.5.1　安装 pytest-cov

pytest-cov 同样可以用 pip 安装，具体如下。

```
$ pip install pytest-cov
```

在本地开发环境中安装完毕后，别忘了在 Jenkins 的 virtualenv 环境中也安装一遍。

10.5.2　从 pytest 获取覆盖率

执行下述命令，输出 xml 格式的覆盖率报告。

```
$ py.test --cov=foo --cov-report=xml
```

报告会以固定名称 coverage.xml 输出到当前目录下。如果想同时输出 xUnit 格式的文件，则要用下述命令。

```
$ py.test --junit-xml=test_result.xml --cov=foo --cov-report=xml
```

然后将 Jenkins 对应 Job 的“Execute shell”设置值替换成上述命令。

10.5.3　读取覆盖率报告

最后，让 Jenkins 读取上述命令输出的 xml 文件并显示在屏幕上即可。

点击“增加构建后操作步骤”按钮，选择“Publish CoberturaCoverage Report”。随后该选项卡内会自动生成一个“Publish Cobertura Coverage Report”表格项。现在点击“高级”按钮，按照下述方法在图 10.9 中进行设置。

- Cobertura xml report pattern：coverage.xml
- Source Encoding：UTF-8

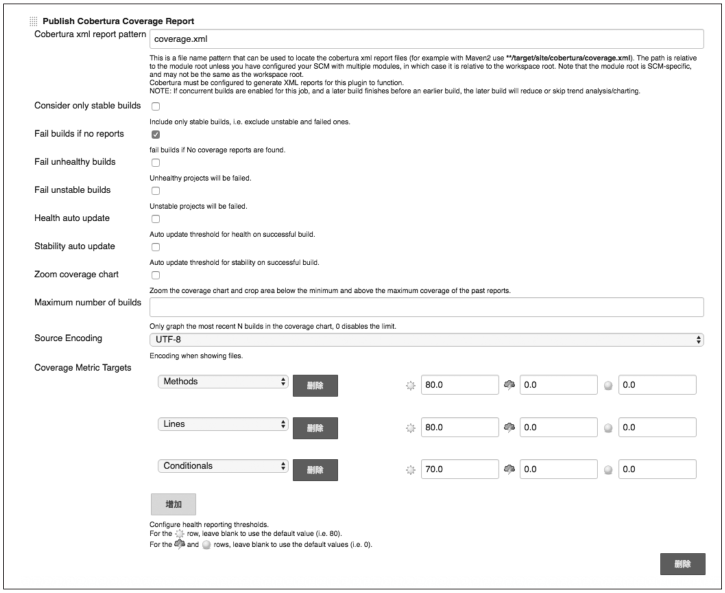


图 10.9　设置读取覆盖率报告

设定完成之后执行 Job，接下来各位应该能在屏幕上看到如图 10.10 所示的覆盖率报告了。

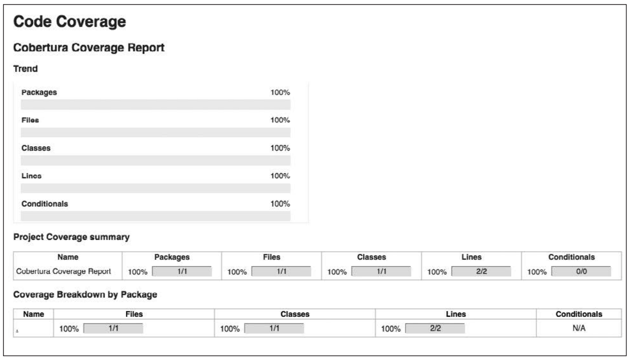


图 10.10　覆盖率报告

测试覆盖到的源码行显示为绿色，反之显示为粉红色。

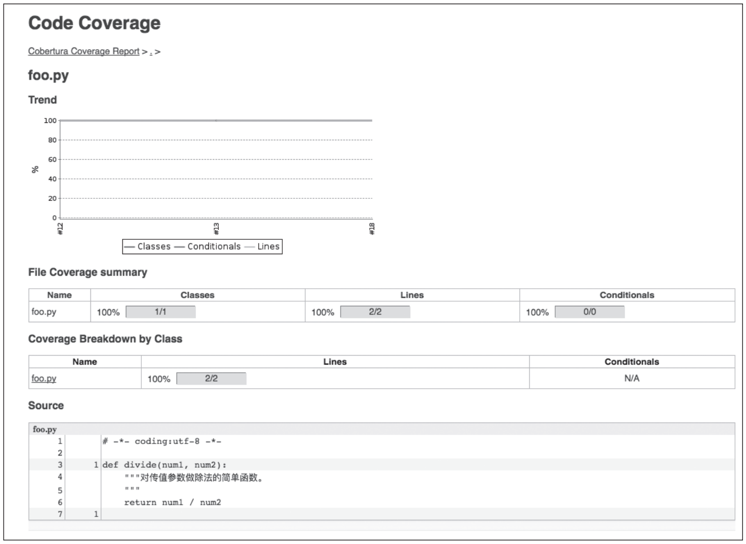


图 10.11　查看各个文件的覆盖率

如图 10.11，这样一来我们就能掌握执行测试代码之后的覆盖率了。接下来我们将学习 Django 测试代码的执行方法。

10.6　执行 Django 的测试

前面我们学习了涉及到 unittest 和 pytest 的测试代码的执行方法，这里我们学习一下如何执行 Django 的测试。Django 有着专用的测试机制，各位在采用 Django 时可以拿本节内容作为参考。在学习如何执行测试的同时，我们还会学习如何向邮箱、Slack 等发送通知。

10.6.1　安装 Python 模块

这里要安装 Django 主体程序以及 Django 与 Jenkins 进行联动时所需的几个工具。

◉ Django 主体程序

首先安装 Django 主体程序。

```
$ pip install django
```

◉ unittest-xml-reporting

我们知道，要想让 Jenkins 显示测试结果报告，需要将测试执行结果以 Junit 格式的 xml 文

件形式输出。

使用 Django 的 `manage.py test` 命令执行测试后，生成的结果只是纯文本，并不满足上述格式要求。用 unittest-xml-reporting 可以解决这一问题，其安装如下。

```
$ pip install unittest-xml-reporting
```

◉ coverage

还要安装 coverage 模块。

```
$ pip install coverage
```

10.6.2　Django 的调整

◉ 创建项目目录

创建项目目录。这里我们用 cafe/apps 作为其名称。

```
$ mkdir cafe
$ cd cafe
$ django-admin.py startproject apps
```

这个项目由 Mercurial 的版本库管理，保存在 /var/hg/cafe 目录下。

◉ 创建 Django 应用

接下来创建 Django 应用。运行上面那条命令之后，我们会得到 apps 目录。现在移动到该目录下，创建一个名为 menu 的应用。

```
$ cd apps
$ python manage.py startapp menu
```

执行完毕后将生成 apps/menu 目录。

10.6.3　示例代码

◉ 编写模型

编写模型。我们这里所用的概念是一间咖啡厅的茶品清单，代码如 LIST 10.4 所示。

LIST 10.4　cafe/apps/menu/models.py

```
# -*- coding:utf-8 -*-
from django.db import models
```

```
TEA_KINDS = (
    ("english", u"英式红茶"),
    ("chinese", u"中国茶"),
    ("japanese", u"日本茶"),
)

class TeaManager(models.Manager):
    def recommended(self):
        """ 仅显示推荐商品 """
        return self.filter(is_flavor=True)

    def count_each_kind(self):
        """ 以字典形式返回各类茶的件数 """
        result = self.values_list("kind").annotate(
                    count=models.Count("kind"))
        return dict(result)

class Tea(models.Model):
    objects = TeaManager()

    name = models.CharField(u"名称", max_length=255)
    kind = models.CharField(u"种类", max_length=255, choices=TEA_KINDS)
    price = models.IntegerField(u"价格")
    is_recommended = models.BooleanField(
                        u"推荐商品", default=False)
```

◉ 编写表单

表单代码如 LIST 10.5 所示。

LIST 10.5　cafe/apps/menu/forms.py

```
# -*- coding:utf-8 -*-
from django import forms

from menu.models import Tea, TEA_KINDS

class TeaSearchForm(forms.Form):
    name = forms.CharField(label=u"", max_length=255, required=False)
    kind = forms.MultipleChoiceField(
                label=u"种类", choices=TEA_KINDS, required=False)
    extra_report = forms.BooleanField(
                label=u"输出追加报告", required=False)
```

```
    def clean(self):
        clnd = self.cleaned_data
        if not self.is_valid():
            return clnd

        if not clnd["name"] and not clnd["kind"]:
            raise forms.ValidationError(
                      u"名称和种类请至少输入一项")

        return clnd
```

◉ 向 settings.INSTALLED_APPS 添加 menu

创建完 menu 应用后记得在 settings.py 的 INSTALLED_APPS 里指定它（LIST 10.6）。缺少这一步是不能用 manage.py 执行测试的。

LIST 10.6　cafe/apps/settings.py

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'menu',
)
```

◉ 为 Jenkins 写 settings

我们希望 Jenkins 利用 unittest-xml-reporting 将测试结果以 xml 文件的形式输出，所以这里要专门为 Jenkins 写一个 settings。

代码如 LIST 10.7 所示。

LIST 10.7　cafe/apps/settings_jenkins.py

```
# -*- coding:utf-8 -*-
from settings import *

######################
# Xml test runner
######################
TEST_RUNNER = 'xmlrunner.extra.djangotestrunner.XMLTestRunner'
```

```
TEST_OUTPUT_DIR = 'test_reports'

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'jenkins.db',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
```

◉ 生成迁移文件

前面我们用示例代码创建了模型，所以这里还需要生成迁移文件。移动到 manage.py 文件所在的目录下执行下述命令，即可完成迁移文件的生成。

```
$ python manage.py makemigrations
```

◉ 编写测试代码

模型和表单的代码都已经写好，接下来该编写测试代码了。测试代码如 LIST 10.8 所示。

LIST 10.8　cafe/apps/menu/tests.py

```
# -*- coding:utf-8 -*-
import unittest
from django.test import TestCase as DjangoTest

from menu.models import Tea
from menu.forms import TeaSearchForm

row = lambda L: (dict(zip(L[0], x)) for x in L[1:])

class TeaManagerTest(DjangoTest):
    def setUp(self):
        datas = (
            ("name", "kind"),
            (u"大吉岭", "english"),
            (u"锡兰红茶", "english"),
            (u"乌龙茶", "chinese"),
            (u"铁观音", "chinese"),
            (u"普洱茶", "chinese"),
            (u"静冈茶", "japanese"),
        )
```

```
        for data in row(datas):
            Tea.objects.create(price=100, **data)

    def test_count_each_kind(self):
        result = Tea.objects.count_each_kind()
        self.assertEqual(result, dict(english=2, chinese=3, japanese=1))

class TeaSearchFormTest(unittest.TestCase):
    def test_valid(self):
        """ 检查输入正常时是否会报错 """
        params = dict(name="foo", kind=["english"])
        form = TeaSearchForm(params)
        self.assertEqual(form.is_valid(), True, form.errors.as_text())

    def test_either1(self):
        """ 检查名称和种类都无输入时是否会报错 """
        params = dict()
        form = TeaSearchForm(params)
        self.assertEqual(form.is_valid(), False, form.errors.as_text())

    def test_either2(self):
        """ 检查输入名称后是否会报错 """
        params = dict(name="foo")
        form = TeaSearchForm(params)
        self.assertEqual(form.is_valid(), True, form.errors.as_text())

    def test_either3(self):
        """ 检查输入种类后是否会报错 """
        params = dict(kind=["english", "chinese"])
        form = TeaSearchForm(params)
        self.assertEqual(form.is_valid(), True, form.errors.as_text())
```

10.6.4　Jenkins 的调整

◉ 新建 Job

点击 Jenkins 面板侧边栏的“新建”，创建 Job。

① 选择“构建一个自由风格的软件项目”

② 在“Item 名称”处输入 cafe（Job 名并没有硬性要求，这里只是特意选择了与项目名称相同的 cafe）

◉“源码管理系统”的设置

这里使用 Mercurial。我们事先已经将 cafe 的版本库放在了 /var/hg/cafe 下，所以直接在

Jenkins 的设置界面作如下输入即可。

- 源码管理系统：Mercurial
- Repository URL ：/var/hg/cafe
- Branch ：default
- 源码库浏览器：Auto

◉ 在 Execute shell 中填写测试命令

把 Django 测试结束后输出覆盖率报告的命令写在“Execute shell”中。如果只是单纯地进行 Django 测试，可以用下述命令执行。

```
$ PYTHONPATH=apps python -m manage test menu --settings=settings_jenkins
```

通过 `coverage` 命令执行上述代码即可获取覆盖率。接下来，我们将上述代码改成下面这样。

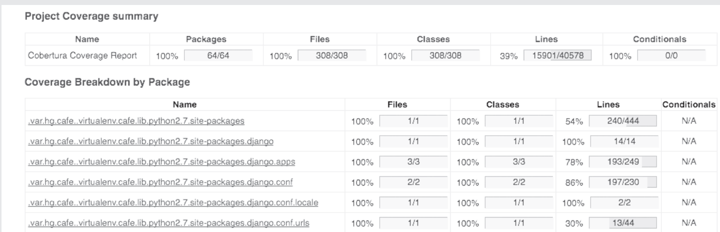
```
$ PYTHONPATH=apps coverage run --source=apps -m manage test menu --settings=s
ettings_jenkins && coverage xml -o coverage_reports/coverage.xml
```

然后，run 成功之后，执行 `coverage xml` 命令即可输出 xml 格式的报告。

这里我们指定了 --source=apps 选项。该选项可以将覆盖率统计对象限制在指定的目录下。在遇到这种 Django 项目目录与版本库路径不一致的情况时，指定该选项能有效防止读取无关代码。

专栏　不要直接在 Django 的项目目录下执行测试

获取覆盖率时有一点需要注意，那就是不能直接在 Django 的项目目录下执行测试。使用 Django 时有很多工作是在 Django 的项目目录下进行（比如上面例子中移动到 apps 目录下并执行 `manage.py test`）。如果用上述方法统计覆盖率，那么一旦我们移动到 apps 并执行 `manage.py test`，会连同 Django 本身一同测试并统计覆盖率，最终的报告会变成图 10.12 这个样子。因此，为了防止非测试对象混入覆盖率报告，必须对这一点强加注意。



Project Coverage summary

Name	Packages	Files	Classes	Lines	Conditionals
Cobertura Coverage Report	100% 64/64	100% 308/308	100% 308/308	39% 15901/40578	100% 0/0

Coverage Breakdown by Package

Name	Files	Classes	Lines	Conditionals
.var.hg.cafe..virtualenv.cafe.lib.python2.7.site-packages	100% 1/1	100% 1/1	54% 240/444	N/A
.var.hg.cafe..virtualenv.cafe.lib.python2.7.site-packages.django	100% 1/1	100% 1/1	100% 14/14	N/A
.var.hg.cafe..virtualenv.cafe.lib.python2.7.site-packages.django.apps	100% 3/3	100% 3/3	78% 193/249	N/A
.var.hg.cafe..virtualenv.cafe.lib.python2.7.site-packages.django.conf	100% 2/2	100% 2/2	86% 197/230	N/A
.var.hg.cafe..virtualenv.cafe.lib.python2.7.site-packages.django.conf.locale	100% 1/1	100% 1/1	100% 2/2	N/A
.var.hg.cafe..virtualenv.cafe.lib.python2.7.site-packages.django.conf.urls	100% 1/1	100% 1/1	30% 13/44	N/A

图 10.12　混入多余代码之后的覆盖率

10.6.5　“构建后操作”选项卡的设置

“构建后操作”选项卡里设置的任务将在 Job 的构建处理执行完毕后执行。

Jenkins 读取测试结果报告与覆盖率报告的操作需要在这里进行设置。另外，执行 Job 失败的邮件通知也是在这里设置的。

◉ 读取测试结果报告

“Publish JUnit test result report”中有“测试报告（XML）”一项，用于指定输出的 xml 文件。

这里允许使用通配符。unittest-xml-reporting 会针对每一个测试类输出一个 xml 文件，所以按照下述方法设置即可。另外，测试结果的输出目录要与 Jenkins 的 settings 中指定的目录一致。

- 测试报告（XML）

  `cafe/test_reports/*.xml`

◉ 读取覆盖率报告

“Publish CoberturaCoverage Report”里有“Cobertura xml report pattern”，我们要在这里指定 coverage xml 命令输出的文件。

- Cobertura xml report pattern

  `cafe/coverage_reports/*.xml`

◉ 查看结果

上述设置完成之后，我们就可以看到测试结果报告和覆盖率报告了，如图 10.13 和图 10.14。

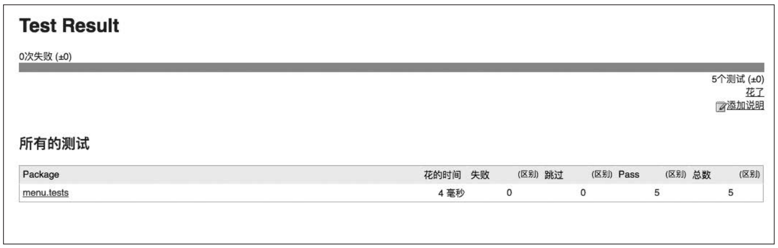


图 10.13　测试结果报告

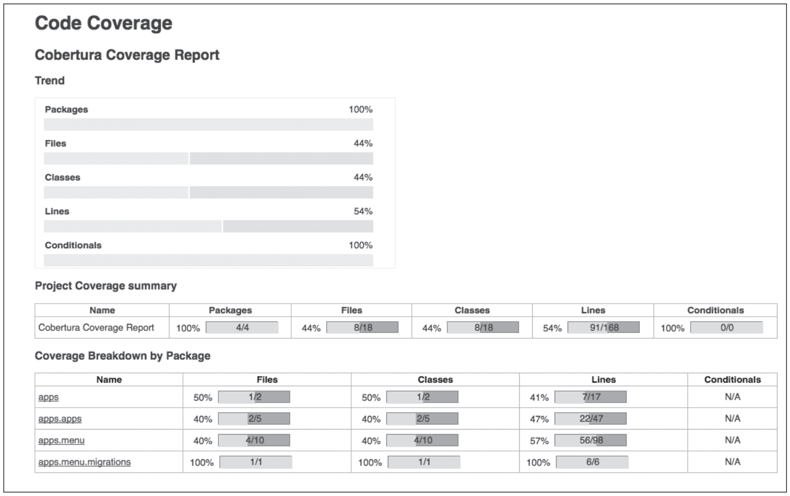


图 10.14　覆盖率报告

NOTE

输出代码覆盖率报告时可能会出现乱码，这是 Java 字体设置导致的。解决中文乱码的方法如下。

・Linux

配置环境变量 export JAVA_TOOL_OPTIONS="-Dfile.encoding=UTF-8"

・Windows

新建变量 JAVA_TOOL_OPTIONS ，value 值为 -Dfile.encoding=UTF-8

重启 Jenkins 之后再执行一遍刚刚出现乱码的 Job，可以看到乱码问题已经被解决。

◉ 邮件通知

“E-mail Notification”中有“Recipients”，我们要在这里指定通知接收方的邮箱地址，如图 10.15 所示。

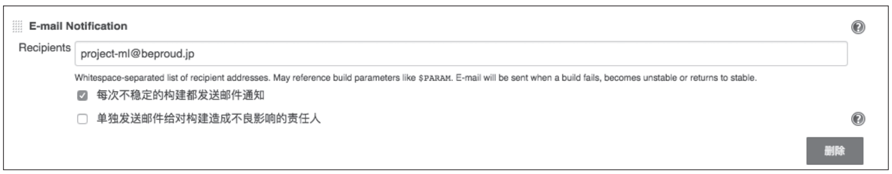


图 10.15　设置收件人

除此之外，还要在侧边栏“系统管理”→“系统设置”里找出“Jenkins Location”，在“系

统管理员邮件地址”中填写 Jenkins 用来发送邮件的邮箱地址，然后在“E-mail Notification”下的“SMTP 服务器”中设置 SMPT 服务器（图 10.16、图 10.17）。另外，如果要使用 Gmail 等需要认证的邮箱账户，需要点击“E-mail Notification”中的“高级”按钮，指定“用户名”“密码”等信息。

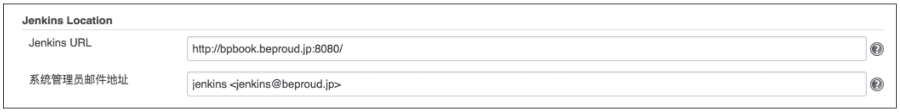


图 10.16　设置发件人的邮箱地址

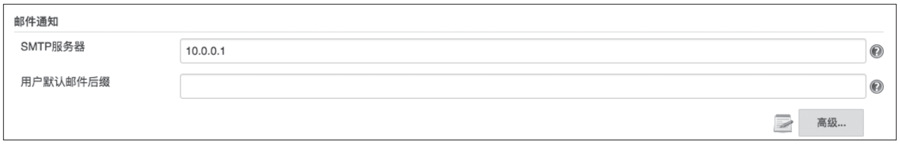


图 10.17　设置发件人的邮箱服务器

在上述设置完成之后，Jenkins 会在 Job 执行失败时发送邮件，以通知开发者。

> **NOTE**
>
> 安装插件后还可以向即将讲到的 Slack、IRC、Jabber、Growl 等发送通知。另外，安装 Email-ext plugin 之后还可以对邮件的主题、正文、通知设置等进行自定义。

◉ 向 Slack 发送通知

Slack 是一款可以跨平台使用的团队交流工具，还可以与许多第三方服务集成。Jenkins 可以借助 Slack Plugin 向 Slack 发送通知。Slack 的相关说明请参考 4.3 节。

首先进行 Slack 的设置。找出需要通知的频道或组，选择“Add a service integration”，添加“Jenkins CI”。接下来会跳转到“Post to Channel”界面，确认要通知的频道或组无误后，点击“Add Jenkins CI Integration”按钮，跳转至自定义界面。这里要先将“Step 3”中写的“TeamDomain”和“Integration Token”复制下来，因为后面设置 Jenkins 时会用到。Slack 发送通知时所用的名字和头像可以在这个界面修改。最后点击“Save Setting”完成 Slack 的设置。

接下来设置 Jenkins。在 Jenkins 的“系统管理”→“系统设置”中找到“Global Slack Notifier Settings”，然后作如下更改（图 10.18）。

① TeamDomain：设置 Slack 时复制的 TeamDomain

② Integration Token：设置 Slack 时复制的 Integration Token

③ Channel：Slack 的频道名或组名

④ Build Server URL：Jenkins 服务器的 URL

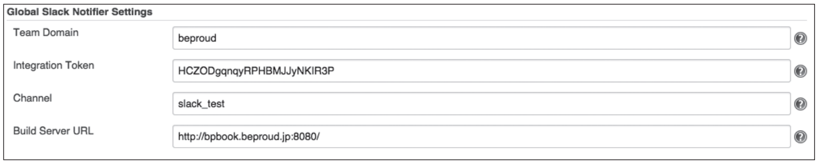


图 10.18　Global Slack Notifier Settings 示例

接着在 Job 的设置中找到“Slack Notifications”，勾选要通知的项目，最后在“构建后操作”处选择添加 Slack Notifications”即可（图 10.19）。



图 10.19　通知项目示例

现在再执行 Job 就会向 Slack 发送通知了（图 10.20）。

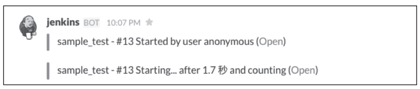


图 10.20　向 Slack 发送通知

10.7　通过 Jenkins 构建文档

我们在第 7 章中介绍了如何用 Sphinx 写文档，现在来看看如何用 Jenkins 来辅助 Sphinx 编写文档。

用 Sphinx 编写文档时经常会遇到下述问题。

- reST 语法错误
- 忘记提交文档中引用的图片或源码

如果存在上述问题，Sphinx 会在执行构建命令时发出警告。因此，只要我们利用 Jenkins 定期监视文档的构建，就能尽早发现这些问题。

本书就是借助 Sphinx 编写的。接下来我们将以本书的编写过程为例，学习一下我们应该怎样用 Jenkins 管理多人共同执笔的书稿。该方法不仅适用于多人共同管理文档的情况，对单独管理文档的情况也有一定的帮助。

10.7.1　安装 Sphinx

首先将 Sphinx 安装到本地环境中。

```
$ pip install sphinx
```

10.7.2　在 Jenkins 添加 Job

该 Job 的设置如下（/var/hg/bpbook 为源码版本库）。

① 基本设置

- Job 类型：自由风格
- Job 名称：bpbook

② 源码管理系统

- SCM：Mercurial
- 版本库：/var/hg/bpbook
- 分支：default

③ 构建触发器

- 种类：定期执行
- 日程表：H/15****（每小时的 15 分）

10.7.3　Sphinx 构建发出警告时令 Job 失败

Sphinx 的 -w 选项可以将警告内容以文件形式输出。我们希望出现警告时让 Jenkins 的 Job 失败，所以要在“Execute shell”选项卡处进行如下设置：当输出警告内容的文件不为空时，用

返回码 9 结束执行。

```
sphinx-build -a -E -w build_warnings.txt source docs

if [ -s build_warnings.txt ] ; then
    exit 9
fi
```

这里的返回码可以用 0 之外的任意数字。Jenkins 将所有构建完毕时返回码不为 0 的 Job 均视为失败。另外，为保证每次都重新构建所有文档，要加上 -a –E 选项。

10.7.4　查看成果

Jenkins 可以在管理界面查看工作区内的文件，因此我们可以在这里查看 Job 构建的文档（图 10.21）。

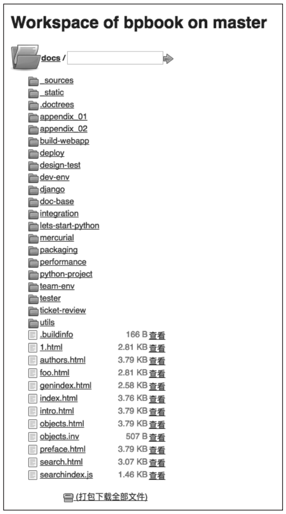


图 10.21　查看工作区

Jenkins 中的各 Job 首页的“描述”中允许添加 html 标签，我们可以将已构建的文档的链接插入“描述”中，这样一来就能直接从首页点击链接查看构建后的文档了（图 10.22）。

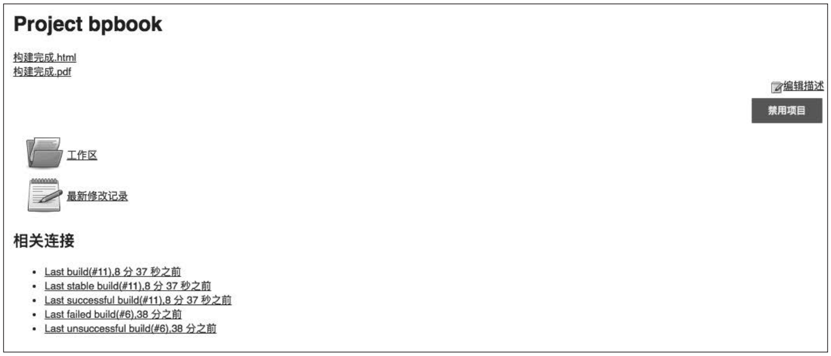


图 10.22　编辑 Job 首页的“描述”

10.7.5　通过 Task Scanner Plugin 管理 TODO

编写代码和稿件时，我们常需要将一些临时想到的事或暂时不需要做的事以 TODO 注释的形式记录下来。用 Jenkins 的 Task Scanner Plugin 插件可以让我们在报告中看到 TODO 注释（图 10.23）。

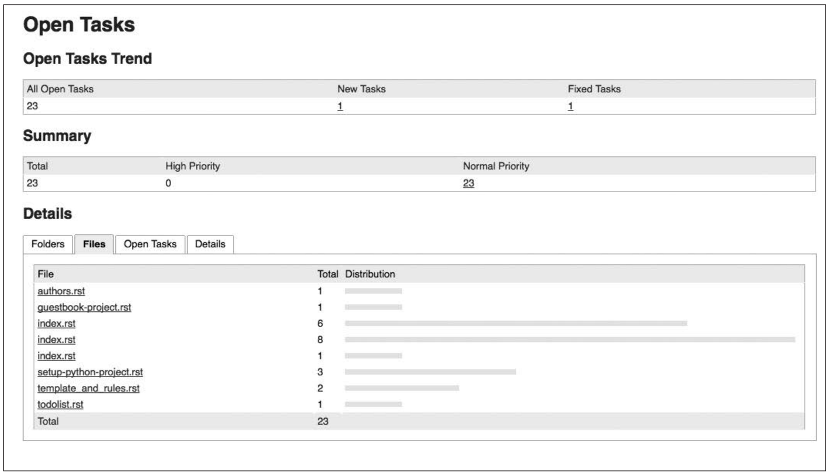


图 10.23　Task Scanner Plugin 报告

如图 10.24 所示，报告中还对对应行做了高亮处理。

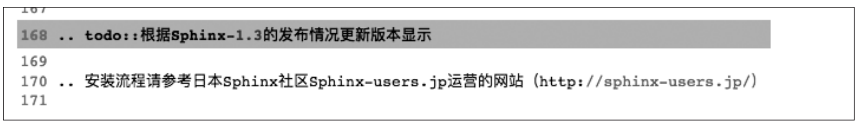


图 10.24　高亮显示

另外，Sphinx 扩展中有一个 todo 扩展，激活之后即可允许用户使用 todo 指令（图 10.25）。todo 指令的布局与 note 和 warning 相同。

Todo: 根据Sphinx-1.3的发布情况更新版本显示

图 10.25　使用 todo 指令的示例

与其他人合著一本书时，我们常会怀疑自己写的内容与其他章节是否存在矛盾，或是自己写的稿子是否满足与合著者之间定下的规则。如果每出现一次这种情况就停下手去检查一遍，势必拖累稿子的进度，所以我们选择先留下 TODO 注释，等待最后统一解决。

另外，TODO 注释经常写了不删，让人搞不清到底问题有没有解决。这种情况不仅存在于写稿的过程中，在敲代码时也同样会发生。虽然每次处理完源码或稿件都手动 grep 搜索 TODO 注释能避免这种情况发生，但实在太麻烦了。

因此，在用 Jenkins 执行测试或构建文档时，可以顺便生成一个 TODO 的列表。另外，Task Scanner Plugin 能够用图表显示 TODO 注释的数量。随着 TODO 被一个个解决，我们能够直观地掌握其减少的过程。看到 TODO 减少能够激发开发者解决 TODO 的积极性，在某种意义上起到提高效率的作用。

10.7.6　Task Scanner Plugin 的设置示例

首先从“管理插件”安装 Task Scanner Plugin。完成后，Job 设置界面的“构建后操作”选项卡处会增加“Scan workspace for open tasks”一项。勾选该项，输入下述几项。

- Files to scan：TODO 的搜索对象
- Files to exclude：这里设置的对象将不包含在搜索范围内
- Tasks tags：这里输入的字符串将被视为 TODO，计入统计结果

管理本书稿的 Job 就是按照图 10.26 设置的。

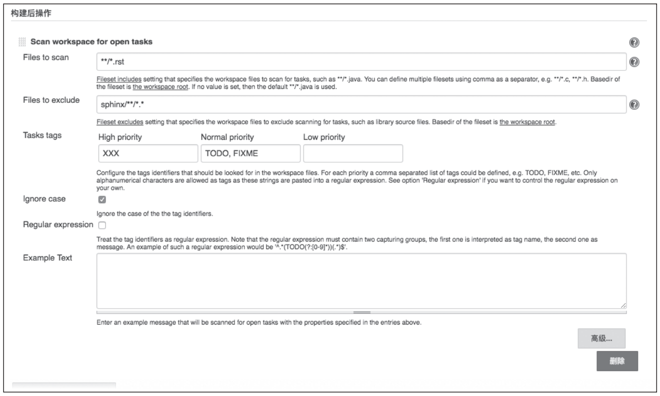


图 10.26　Task Scanner Plugin 设置示例

10.8　Jenkins 进阶技巧

10.8.1　好用的功能

这里简单了解一下前面没有讲到的 Jenkins 的好用功能。

◉ Job 触发器

Job 触发器是指通过 Jenkins 的某个 Job 来启动其他 Job 的功能。它没有专门的设置界面，必须到“构建触发器”的“Build after other projects are built”或“构建后操作”的“Build other projects”中进行设置。

这里可以设置前一个 Job 失败时不执行后续 Job，因此能够构成“测试不通过则不进行部署”的工作流程。

◉ 用户管理

用户管理可以设置 Jenkins 的认证与认可。设置可以满足很多细节，比如只有公司内部人员可通过外部网络访问 Jenkins，或者只有管理员能够执行 Job，等等。

另外，用户管理后端可以使用 LDAP，因此可以将用户管理交给 OpenLDAP 或 Active Directory。使用 Active Directory 时建议安装 Active Directory plugin，它能够让我们轻轻松松地进行设置。

> **NOTE**
>
> 我们可以在下列 URL 上找到关于 Jenkins 用户管理的详细介绍，有兴趣的读者不妨参考一下。
>
> Standard Security Setup
> `https://wiki.jenkins-ci.org/display/JENKINS/Standard+Security+Setup`
>
> Active Directory plugin
> `https://wiki.jenkins-ci.org/display/JENKINS/Active+Directory+plugin`

◉ 集群

Jenkins 允许以主从方式组建集群。我们在使用 Jenkins 时，往往一建就是十几二十个定期执行的 Job。另外，Job 的执行时间可能会被测试拖长（测试过慢），导致 1 台计算机无法处理所有 Job。这种时候组件集群就显得行之有效了。此外，对于需要在多个不同环境中进行的测试而言，集群也是一种很好的解决方法。

简单来说，Jenkins 集群的机制如下所述。

- 主计算机向从计算机发送指令
- 从计算机执行指令
- 主计算机统计从计算机的结果

> **NOTE**
>
> 我们可以在下列 URL 上找到 Jenkins 集群的详细介绍，有兴趣的读者不妨参考一下。
>
> Distributed builds
> `https://wiki.jenkins-ci.org/display/JA/Distributed+builds`

◉ CLI

我们在 10.2.2 节了解过，除了 Web UI 以外，Jenkins 还有其他 CLI。Jenkins 的 CLI 很适合用来执行插件定期升级等标准化处理。另外，Jenkins 的 CLI 允许远程执行，便于用户在不能使用 Web 浏览器的终端上操作 Jenkins。

NOTE

我们可以在下列 URL 上找到 Jenkins CLI 的详细介绍，有兴趣的读者不妨参考一下。

Jenkins CLI

https://wiki.jenkins-ci.org/display/JA/Jenkins+CLI

10.8.2 进一步改善

到 10.7 节为止，我们学习了如何用 Jenkins 签出源码、执行测试（测量覆盖率）、统计 TODO 任务、发送结果通知和编写文档等。但是，这些内容只是整个开发流程的一部分。相信各位还希望能用 Jenkins 做更多的事，比如下面这些。

- Pylint 等的静态解析
- 向正式环境或演示环境部署
- 测试已部署的产品
- 向 PyPI 服务器上传
- 所交付产品的 Zip 文件存档

另外，只要 Jenkins 的运用步入正轨，我们将会在改善业务流程、根据业务流程自定义 Jenkins 等方面产生需求。

遇到此类需求时可以参考下述信息。

NOTE

官方维基百科的信息

- Jenkins Home: https://wiki.jenkins-ci.org/display/JENKINS/Home

邮件列表

- http://jenkins-ci.org/content/mailing-lists

书籍

- 《Jenkins 入门与实践》[①]（技术评论社，2011 年）
- *Jenkins: The Definitive Guide*[②]（O'Reilly Media，2011 年）
- 《持续集成：软件质量改进和风险降低之道》（机械工业出版社，2008 年）

---

① 原书名为『Jenkins 実践入門』，暂无中文版。——译者注

② 本书作者为 John Ferguson Smart，暂无中文版。——译者注

·《Jenkins 入门》[①]（Shuwa System，2012 年）

虽然用 Jenkins 能完成许多精细的工作，但这不代表自动化优于一切。使用 Jenkins 实现自动化时，务必要保证作业成本和便捷性的平衡。

10.9　小结

本章对持续集成进行了简单的说明，同时讲解了 Jenkins 的用法。读完本章之后，相信各位已经发现持续集成和 Jenkins 并没有想象中那么难。虽然本章主要是以 Django 为例进行讲解的，但 Jenkins 的用武之地远不止于此。各位可以从小规模项目入手，逐步感受导入 Jenkins 的方便之处。

① 原书名为『入門 Jenkins』，暂无中文版。——译者注

第 3 部分

服务公开

第 3 部分主要讨论与正式环境相关的一些较现实的问题。这部分将介绍向正式环境公开项目时的思路和流程，研究在正式环境中改善性能的相关问题。

第 11 章　环境搭建与部署的自动化

应用的运行离不开执行环境。

搭建环境要做的事非常多，比如安装依赖包、设置中间件、设置应用本身等。这些工作稍有差错就会导致应用无法运行，而我们很难分辨问题究竟出在应用身上还是环境身上，因此解决问题常常要费一番力气。

另外，许多项目的环境不止一个，除正式环境外还需要开发环境、演示环境等。现今还要考虑到向外扩展，有时一个环境甚至是由几十台服务器共同构成的。要手动搭建这么多环境还要保证不出错，实在有些强人所难。

为保证良好的服务运营，需要有能让应用稳定运行且便于维护的环境。本章将就搭建稳定环境的方法理论以及通过 Ansible 实现环境搭建、部署自动化的方法进行说明。

本章的目的是搭建环境的自动化，但是如果过于随意地搭建内容和流程，则很难形成一个稳定的环境。另外，在这种状态下也不可能顺利实现自动化。因此本章内容将分为两部分，第一部分探讨搭建环境的流程和内容，第二部分讲解如何将第一部分的内容自动化。

11.1　确定所需环境的内容

首先要从所需环境的内容入手。环境一旦开始使用就很难再进行大的改动，所以动手搭建前搞清所需环境的内容是十分重要的。

在熟悉搭建环境的流程之前，建议先摸索着尝试手动搭建。等到熟悉之后，再考虑结构选择和搭建过程自动化的问题。

利用 VirtualBox 和快照功能可以帮助我们快速确定所需环境的内容。感兴趣的读者请同时参考附录 A。

11.1.1　网络结构

许多服务是由多台服务器组合实现的，所以在探讨服务器内部的环境搭建之前，先要确定服务器的组成结构。

可用服务器数、预算、应用性能不同，所需的服务器结构也是千差万别。这里我们以图 11.1 所示的结构为例进行学习。

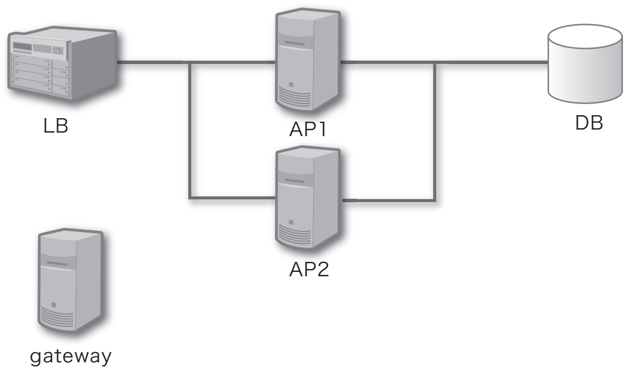


图 11.1　服务器结构

- LB：负载平衡器（nginx）
- AP1、2：应用服务器（django/gunicorn）
- DB：数据库（mysql）
- gateway：跳板服务器

◉ 服务器编组

确定服务器结构时，要给职责相同的服务器起一个统一的名字。

这里我们将分担各个职责的服务器群称为组。在图 11.1 所示的结构中，有 LB、AP、DB、gateway 这 4 个组。

即便某个职责只由一台服务器完成，我们也认为其是一个组。这样一来能更灵活地应对多服务器结构。

> **NOTE**
>
> 按职责划分的服务器群也被称为“角色”（Role）。但这会与我们即将讲到的 Ansible 的 Role 重复，所以这里改用“组”（Group）。

◉ 跳板服务器

允许从外部网络直接 ssh 登录各个服务器会带来很多安全隐患，所以一般我们会阻止外部通过 ssh 访问各服务器。

可是，一旦阻止了对所有服务器的 ssh，我们便无法通过外网登录，这会造成很多不便。所以这里要准备一台用作登录跳板的服务器，形成通过跳板登录各服务器的结构。

11.1.2　服务器搭建内容的结构化

将服务器搭建内容按照图 11.2 所示的结构进行结构化。

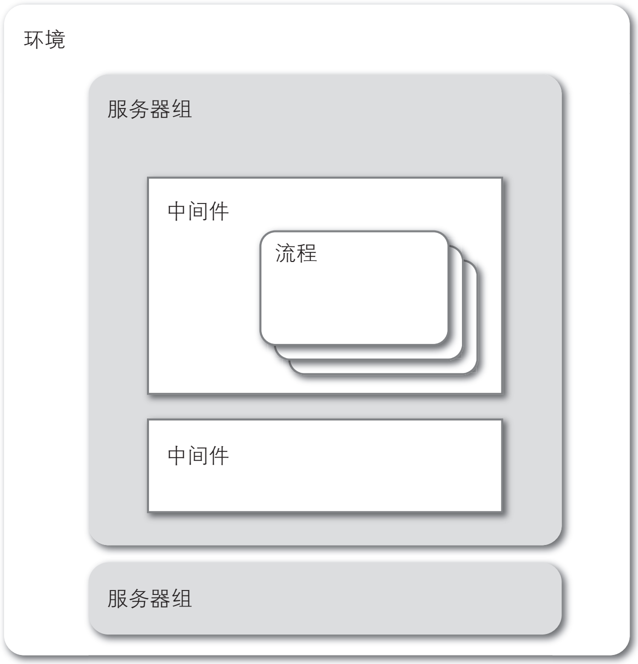


图 11.2　服务器搭建内容的结构化

这个结构很好理解。在中间件里定义安装和设置等流程，然后将各个中间件组合起来搭建服务器组。最后把搭建好的多个服务器组整合起来，就形成了一个环境。

这里需要特别注意的是，将搭建流程整合到各个中间件里时，如果只简单地把要做的处理按顺序一件一件写下来，很容易让人搞不清究竟哪个步骤属于哪个中间件的设置。加强对这个结构的理解和注意，有助于提高自动化的效率。关于自动化的知识我们将在 11.2 节进行学习。

根据图 11.2 所示的结构，本章中的服务器可划分为如下图 11.3 所示的结构。



图 11.3　本章中的服务器的搭建内容

这里出现了“环境设置”，它不是中间件，而是我们为了便于理解，将 OS 的设置、用户的创建等不针对特定中间件进行的操作汇总在了一起，并统一称为“环境设置”。

下面我们开始学习具体的搭建流程。

11.1.3　用户的设置

我们已经确定好了服务器的结构，接下来要探讨搭建流程。首先是用户的设置。

初始状态下的 Ubuntu 以 ubuntu 用户为登录用户。但是，如果让默认的管理员用户来做维护或启动应用，会带来一些安全隐患。所以，我们新建 mainte 用户用于搭建和维护，新建一个不能直接登录的普通用户 www 用于执行应用。

```
# useradd -m mainte
# useradd -m www
```

搭建工作会经常用到 sudo，因此要给 mainte 用户设置 sudo 权限。另外，考虑到自动化的问题，最好让 mainte 不需要密码就可以执行 sudo。

可以通过编辑 /etc/sudoers 或者在 /etc/sudoers.d/ 中添加设置文件修改 sudo 权限。这次我们用后一种方法，创建 /etc/sudoers.d/mainte。

```
%mainte ALL=(ALL) NOPASSWD:ALL
```

%mainte 表示给 mainte 组的所有用户设置 sudo 权限。今后如果有其他用户需要用到 sudo，只要将其加入 mainte 组即可。

至于 www 用户，要让它只保有启动应用所需的最小权限，因此不设置 sudo 权限。

最后是登录设置。mainte 用户今后要用来做维护工作，所以需要能从外部登录。

首先用 ssh-keygen 生成密钥文件。

```
$ sudo su - mainte
$ ssh-keygen -b 2048
```

如果在初始设置状态下执行 ssh-keygen，会在 /home/mainte/.ssh/ 下生成 id_rsa 和 id_rsa.pub 两个文件。我们将这两个文件下载到本地 PC 保管。

将 id_rsa.pub 的内容设置到 authorized_keys 中，登录设置就完成了。

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys
```

经过上面的设置，我们在 mainte 用户的“.ssh”目录下生成了密钥文件，同时完成了对应的 authorized_key 的设置。

这样设置下来之后，就能很轻松地在多个服务器之间切换登录了。

最后查看设置是否正确。

- mainte 用户能用下载到本地 PC 的密钥文件登录服务器
- mainte 用户能通过 sudo 切换为 www 用户
- 可以对 localhost 执行 ssh

```
$ ssh localhost
```

11.1.4　选定程序包

接下来安装应用运行所需的程序包。

在 Ubuntu 上运行 Python 应用时，主要通过 apt-get 和 pip 安装程序包。

选择要安装的程序包时请注意以下几点。

① 安装的程序包要尽量少

尽量不要导入与运行应用无关的程序包。

盲目导入程序包会使我们无法准确掌握应用的运行条件，出问题时很难分辨问题出在环境上还是应用本身上。

要导入与运行应用没有直接关系的程序包时，必须先仔细考虑其用途和安装范围，再确定是否要将其加入搭建流程。

下面是一些做判断的例子。

- 应用的目录结构很复杂，排查 Bug 时需要 `tree` 命令的辅助
  任何环境都难免需要排查 Bug，因此需要将 `tree` 命令加入构建流程。
- 想用自己常用的编辑器 emacs
  基本只会在开发环境中用到，所以仅在开发环境中导入。

② 掌握并管理程序包的版本

对开发力度较大的应用 / 库而言，常有从某个版本起发生大幅度规格变更，或是不兼容旧版本设置文件的情况发生。

这种时候如果盲目升级了版本，很容易导致应用无法运行。

要防止这类因版本导致的事故，重点在于把握安装的程序包的版本以及确定更新原则。

原则主要有下面几种，各位可以为每个程序包分别选择合适的原则。

- 完全固定
  不考虑版本升级，仅使用指定版本的程序包。
- 固定至次版本号
  固定主版本号和次版本号，允许加入安全更新和 Bug 修复。

◉ 通过 apt-get 安装程序包

用 apt-get 或 yum 从各个 Linux 发行版的程序包版本库安装程序包时，首先要确认各个发行版的更新原则。

本书所用的 Ubuntu 原则上只加入安全更新和 Bug 修复，不进行其他版本升级。

也就是说，环境本身就处于上述“固定至次版本号”原则的状态，我们不必多作修改。程序包升级时只会加入重大的 Bug 修复，不会更新主版本号和次版本号。

上述原则只要能满足我们的需求即可，没有什么特别需要注意的地方。现在我们可以通过简单的命令安装程序包，具体如下。

```
$ sudo apt-get install packagename
```

如果需要固定版本，则需要在程序包名称后面添加等号以及版本号。具体如下所示。

```
$ sudo apt-get install packagename=1.2.3-4ubuntu3
```

包名和版本号都可以用正则表达式。比如，如果只想固定到次版本号，则上述命令可以写成“1.2\..*”的形式。

◉ 通过 pip 安装程序包

用 pip 安装程序包的流程在第 9 章中有详细介绍，不清楚的读者请参考该部分。

根据 11.1.4 节得出的结果编写 requirements.txt，修正版本指定，依照自身情况选择是否分离一部分到 dev-requires.txt 和 tests-require.txt。

完成 requirements.txt 之后，只需用 pip install -r requirements.txt 进行安装即可。

◉ 封闭环境中的安装

为没有网络连接的服务器搭建环境时，apt-get 和 pip 这种通过外部连接安装程序包的方法就不好用了。另外，程序包提供方（apt 版本库、PyPI、GitHub 等）故障或维护时也是同样道理。

为应对这种情况，我们可以将所需的程序包事先保存在版本库中，然后进行离线安装。我们把这些打包在一起的程序包叫作 bundle。

bundle 不但可以让我们离线安装程序包，还能有效固定版本以及削减搭建时间。

○ 通过 apt-get 安装 bundle

apt-get 安装的程序包都是以“.deb”格式发布的。用 `apt-get download` 命令可以获取我们需要的 deb 程序包。

```
$ mkdir aptcache && cd aptcache
$ sudo apt-get download python3.4
$ ls
python3.4_3.4.0-2ubuntu1_amd64.deb
```

deb 程序包用 `dpkg -i` 命令安装。

```
$ sudo dpkg -i python3.4_3.4.0-2ubuntu1_amd64.deb
```

但是有一个问题，`apt-get download` 命令只能获取目标程序包，无法同时获取该程序包的依赖包。因此要用 `apt-cache depends` 命令查看依赖包，然后在通过上述方法获取它们的 deb 程序包。

```
$ apt-cache depends python3.4
python3.4
  Depends: python3.4-minimal
  Depends: libpython3.4-stdlib
  Depends: mime-support
  Suggests: python3.4-doc
  Suggests: binutils
```

有时这些依赖包本身又依赖于其他程序包，如此循环下去不知何时是个头，所以我们需要一个方法来一次性获取所有相关的程序包。

实际上，用 `apt-get install` 命令安装程序包时，目标程序包及其所有依赖包的 deb 文件全都会被下载到缓存目录下。只要通过 -o 选项临时变更缓存目录，就能将目标程序包和其所有依赖包保存到任意目录下了。

```
$ sudo apt-get install python3.4 -o Dir::Cache::Archives=/home/mainte/aptcache
```

然而有一点需要注意，那就是这个方法无法获取已经安装的程序包，因此需要在初始状态的环境下（比如刚用 VM 搭建完毕的环境）进行操作。

○ 通过 pip 安装 bundle

旧版本的 pip 可以用 pip bundle 命令进行安装，但这个命令在 pip 1.5 被删除。1.5 之后开始使用 wheel。9.1.4 节详细介绍了 wheel 的相关内容，有需要的读者请参考该部分。

将所需程序包全部转换为 wheel。此时也通过 requirements.txt 指定程序包。

```
$ pip wheel -r requirments.txt
```

这个操作会在当前目录的 wheelhouse 目录下生成 wheel。我们将整个 wheelhouse 目录都添加到应用的版本库中。

在各环境下都可以通过下述命令从 wheelhouse 安装程序包。

```
$ pip install --no-index -f /path/to/my/repository/wheelhouse -r requirements.txt
```

○ bundle 的维护

更新或添加新的程序包时，需要将程序包重新打包成 bundle。一旦漏掉这个步骤，应用可能出现在某些环境下能运行，在某些环境下却不能运行的情况，问题很难排查。

重新打包 bundle 是一件非常繁琐的工作，对于 deb 程序包来说尤其如此。开发时会频繁出现程序包的添加和更新，因此过早地 bundle 化会带来许多麻烦。毕竟开发环境很少遇到无法连接外部网络的情况，所以开发中建议使用 apt-get 和 pip install 来安装程序包。等到正式环境就绪再考虑 bundle 化也不迟。

程序包管理进入 bundle 阶段后，程序包的 bundle 化也可以交给构建服务器实现自动化。

11.1.5　中间件的设置

mysql、nginx 等中间件要根据使用环境进行设置。

我们往往需要在大量服务器上实施或更新中间件的设置，如果这些全都手动去完成，很容易出现疏漏，导致发生问题。因此，实现中间件设置的自动化才是上上之选。首先搞清需要自动化的内容，选出对象文件以及要修改的项目。本章用作例子的服务器结构中包含了 mysql、

nginx 和 gunicorn，这里我们就来探讨一下这 3 个中间件的设置。

◉ 让中间件设置生效的方法

自动化更新中间件的设置文件需要以下步骤。

① 在版本库中管理对象文件
② 用模板语言重新描述可变部分
③ 将模板化的文件翻译过来直接覆盖原设置文件

步骤②和③提到的模板功能是 Ansible 标配的功能，关于 Ansible 的知识会在后面讲到。探讨设置内容的过程中需要我们手动改写设置文件，但最终所有设置都要通过上述步骤反映出来。Ansible 采用 Jinja2 为模板编辑器，因此本节的模板文件都是以 Jinja2 格式描述的。

下面我们来看看用 Jinja2 格式描述设置文件可变部分的例子。

◉ mysql

MySQL 的设置描述在"/etc/mysql/conf.d/*"中。本例的设置文件为 /etc/mysql/conf.d/myproject.cnf。

讨论所需项目并编写文件。

```
[mysqld]

bind_address = {{ MY_LOCAL_IP }}
innodb_file_per_table = yes
innodb_buffer_pool_size = {{ MYSQL_INNODB_BUFFER_POOL_SIZE }}
```

在 mysql 的设置中，有些值会根据环境变化，比如根据服务器 IP 变化的 bind_address，根据服务器内存容量变更最优值的 innodb_buffer_pool_size 等。如果事先将这些值设置为变量，可以在中间件变更所在服务器时轻松完成设置更新。

◉ nginx

nginx 的设置描述在 /etc/nginx/conf.d/ap.conf 中。

nginx 的 conf.d 目录下设置有 default.conf 和 example_ssl.conf 两个文件，它们是用来显示 nginx 默认页的，与我们要搭建环境的服务器无关。因此要在探讨设置之前把这两个文件删除。

LIST 11.1　/etc/nginx/conf.d/ap.conf

```
upstream app {
    {% for server in NGINX_UPSTREAMS %}
    server {{ server }}:8000;
    {% endfor %}
}
```

```
server {
    listen {% if SSL %}443{% else %}80{% endif %};
    server_name {{ DOMAIN }};

    {% if SSL %}
    ssl on;
    ssl_certificate {{ SSL.CERTIFICATE }};
    ssl_certificate_key {{ SSL.KEY }};
    {% endif %}

    ...
}
```

需要在 upstream 指令中指定多个反向代理对象的服务器地址。这里我们是用变量 NGINX_UPSTREAMS 来存储服务器地址清单，并通过 for 语句来指定服务器地址的。

另外，我们还对 server 指令内是否存在 SSL 的设置进行了判断，并根据情况进行了 listen 端口的切换以及 ssl_certificate 的设置。出于预算原因，我们的项目并不是每个环境都使用 SSL，所以在这里保证了两种设定之间的简单切换。

可见，利用 Jinja2 模板的 if 和 for 能做出复杂的情况分类（LIST 11.1）。但有一点需要注意，过度使用控制语句会降低程序的可读性，增加修改的难度。

◉ gunicorn

让 gunicorn 通过 upstart 启动。创建 /ect/init/myapp.conf 文件用作启动脚本，脚本内容如下。

> **upstart 的文档**
> http://upstart.ubuntu.com/

```
description "myapp"

start on (filesystem)
stop on runlevel [016]

respawn
console log
setuid www
setgid www
chdir {{ SOURCE_DIR }}/src

exec DJANGO_SETTINGS_MODULE={{ DJANGO_SETTINGS }} {{ SOURCE_DIR }}.venv/bin/
gunicorn myapp.wsgi:application --bind={{ MY_LOCAL_IP }}:8000 --workers={{
GUNICORN_WORKERS }}
```

上述例子将许多值替换为了变量。等到自动化的时候，这些地方就会发挥出其效果了。

- SOURCE_DIR
  放置源码目录。虽然这个值很少因为环境而变，但换成变量能防止键入错误。
- DJANGO_SETTINGS
  Django的settings文件需要根据环境而变。将它设置为变量之后，我们就可以在所有环境中应用同一个模板了。
- MY_LOCAL_IP
  自身服务器的本地IP。直接拿mysql设置文件中定义的变量来用。
- GUNICORN_WORKERS
  gunicorn的worker进程数。

将中间件的设置文件模板化时，需要考虑以下几个问题。

- 哪个设置值需要用变量替换
  因环境而变的值，以及容易输入错的值（比如较长的文件路径）。
- 有多个中间件共用的变量吗
  所有环境共享的值要设计成共用一个变量。

11.1.6　部署

部署就是将应用安置到环境中，使其进入可运行状态。

应用源码的安置、中间件设置的反映、中间件的重启等都属于部署工作。

本节将对上述 3 个工作进行说明。

◉ 源码的安置与更新

源码的安置用 Mercurial 的 clone 和 pull 即可轻松完成。利用标签和分支的功能，还能灵活应对回滚等操作。

如果有从各环境均可连接的公共密钥认证的中央版本库，那么最好从该版本库进行 clone 操作。这里我们假设中央版本库放在 myrepository.com。

clone 时用 11.1.3 节创建的 www 用户。访问版本库时用到的公共密钥和私有密钥要事先设置好。

本例中，我们把版本库设置在 /var/www/myproject。

```
$ cd /var/www/
$ hg clone ssh://myrepository.com/myproject
```

执行 clone 操作之后就完成了应用源码的安置工作。更新工作只需要在各个服务器上对版本库进行 pull/update 即可。

```
$ cd /var/www/myproject
$ hg pull; hg update default
```

根据用途和环境不同可以将 update 位置替换为其他分支名或标签名，以灵活应对各种需求。实现自动化时只需将其改为变量，让其能够被替换即可。

专栏　没有可通过公共密钥认证的中央版本库时该怎么办

源码在 Mercurial 上管理时，只要我们能使用 clone 操作，就能将源码部署到环境中。但是如果中央版本库要求密码认证，那么每次 clone 都要输入密码，不利于自动化。

这种时候可以先将主版本库 clone 到 gateway 服务器上，然后各环境服务器再从 gateway 服务器上的版本库进行 clone，这样就能将输入密码的次数降到最少（图 11.4）。

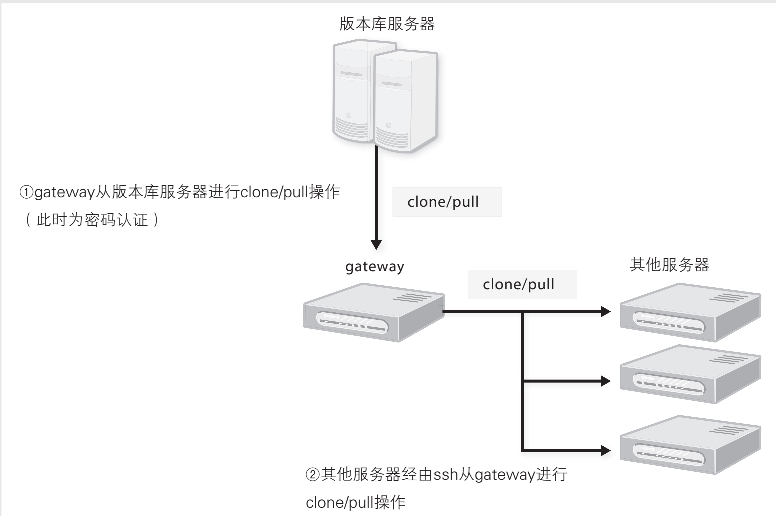


图 11.4　密码认证环境中的源码部署

◉ 中间件设置的反映

如 11.1.5 节所述，现阶段需要手动改写模板部分并上传至各环境，从而反映设置。

改写过程中要时刻注意设置文件有没有放错地方，有没有不小心改写了未加入管理的文件。

◉ 重启中间件

守护进程的启动和停止要使用 `service` 命令。

```
$ sudo service myapp restart
```

11.2　用 Ansible 实现自动化作业

上面我们大致总结了搭建环境的流程，现在来看看如何用 Ansible 实现自动化搭建环境。

11.2.1　Ansible 简介

Ansible 是基于 Python 研发的结构管理工具。不过它除了能进行结构管理之外，还能将许多针对服务器的操作自动化。

Ansible 本身由 Python 实现，但运行所需的设置文件均以 INI 格式或 YAML 格式描述，因此没有 Python 知识也能使用它。

另外，Ansible 不像 Chef 和 Puppet，它不需要在被操作的服务器上安装代理程序。只要服务器允许 ssh 登录，Ansible 就能执行相关操作。

Ansible 的主要概念有以下 5 个，我们随后将依次进行了解。

- inventory
- module
- role
- playbook
- vars

> **NOTE**
>
> 本书只介绍了 Ansible 的一部分功能，想了解其基本使用方法以及其他功能的读者可以参考 Ansible 的官方文档。
>
> `http://docs.ansible.com/`

◉ inventory

inventory 是保存有执行对象（即主机）清单的 INI 格式的设置文件。主机通过 IP 地址或主机名指定。

```
192.168.0.1
192.168.0.2
```

```
192.168.0.3
192.168.0.4
192.168.0.5
```

Ansible 只能访问这里指定的主机，因此我们需要为不同的执行环境准备不同的 inventory 文件。

另外，主机可以用段进行分组。后面讲到的 playbook 和 vars 会用到这里定义的组。

```
[load-balancers]
192.168.0.2

[app-servers]
192.168.0.3
192.168.0.4

[db-servers]
192.168.0.5
```

同一个主机可以同时出现在多个组里。比如在所有功能全由一台主机实现的开发环境中，inventory 就是下面这个样子。

```
[load-balancers]
192.168.0.6

[app-servers]
192.168.0.6

[db-servers]
192.168.0.6
```

Inventory - Ansible Documentation
　http://docs.ansible.com/intro_inventory.html

专栏　Dynamic Inventory

inventory 文件除了可以指定 INI 格式的文件外，还可以指定可执行的脚本文件。我们将这类文件称为 Dynamic Inventory，适用于 Amazon EC2、Google Compute Engine、Docker 等需要频繁变更对象主机信息的情况。

Ansible 的版本库中有以上述 EC2 等为对象的示例 Dynamic Inventory，各位不妨加以参考。

Ansible plugins/inventory
https://github.com/ansible/ansible/tree/devel/contrib/inventory
Dynamic Inventory
http://docs.ansible.com/intro_dynamic_inventory.html

◉ module

Ansible 以 module 为单位定义对服务器的操作。

Ansible 本身自带了许多 module，可以将它们组合起来，实现多种操作。

Module Index - Ansible Documentation
http://docs.ansible.com/modules_by_category.html

只要遵循一定的规则，module 可用任何语言来实现。当标准模块无法满足需求时，可以直接将现有脚本封装成 module 来使用。

Developing Modules - Ansible Documentation
http://docs.ansible.com/developing_modules.html

执行 module 时以 task 形式描述自身的传值参数及其他参数。

```
- name: install nginx
  sudo: yes
  apt: name=nginx
- name: install mysql
  sudo: yes
  apt:
    name: mysql-server
    state: latest
    update_cache: yes
```

module 的传值参数以 key=value 的形式描述，各传值参数之间用空格区分。或者也可以用字典形式描述。传值参数较多时建议使用字典形式。

◉ role

role 可以批量重复利用 task。

role 的结构如 LIST 11.2 所示。

LIST 11.2　role 的目录结构

```
roles/
```

```
+-- nginx/
     +-- tasks/
          +-- main.yml
     +-- handlers/
          +-- main.yml
     +-- vars/
          +-- main.yml
     +-- defaults/
          +-- main.yml
     +-- files/
          +-- nginx.repo
     +-- templates/
          +-- conf.d/
               +-- ap.conf
     +-- meta/
          +-- main.yml
```

- tasks

  task定义。定义描述在该目录的main.yml中。
- handlers

  handler的定义。由task定义的notify指定并调用。同上，在main.yml中描述。
- vars

  该role使用的变量。同上，在main.yml中描述。
- defaults

  上述变量的默认值。同上，在main.yml中描述。
- files

  该role的task中，文件关联模块用到的文件。
- templates

  该role的task中，template模块用到的Jinja2模板文件。
- meta

  元信息。可定义role之间的依赖关系等。

专栏　role 的共享

role 也能像 PyPI 一样在 Web 上公开及共享。Ansible Galaxy[①] 是 Ansible 公司运营的网站，任何人都可以在这里免费上传和下载 role。

① https://galaxy.ansible.com/

从 Ansible Galaxy 上下载 role 时，需要用 ansible-galaxy install 命令。这个命令只需安装 Ansible 就可以使用了。下载的 role 默认安装到 /etc/ansible/roles 目录下，我们可以通过 -p 选项指定安装位置。

```
$ ansible-galaxy install username.rolename -p ROLES_PATH
```

使用 ansible-galaxy init 命令可以生成包含 meta/main.yml 和 tasks 目录等内容的样板文件。这个命令原本是为方便用户向 Ansible Galaxy 上传 role 而设计的，但对于不想公开的 role 同样好用。因此各位在制作 role 的时候不妨试试这个命令。

```
$ ansible-galaxy init rolename
```

◉ playbook

playbook 是 YAML 格式的文件，用来定义要执行的处理。

```
- hosts: load-balancers
  sudo_user: mainte
  roles:
    - django
```

在 hosts 处指定对象服务器。这里指定 inventory 中定义的群名，可以对该群中的主机执行 task。指定为 all 则以所有主机为对象。

定义 sudo: yes 之后，该 playbook 将全部由 sudo 用户执行。sudo 用户默认为 root。想使用 root 以外的用户时需要在 sudo_user 处指定。

roles 处以 YAML 的列表形式指定要执行的 role。虽然可以在 tasks 处直接描述 task，但除了没有现成脚本的情况以外，还是建议使用 role。

> Playbooks - Ansible Documentation
> `http://docs.ansible.com/playbooks.html`

◉ vars

playbook、task、role 的模板中都可以使用变量。vars 的设置方法有很多种，这里只介绍比较有代表性的。

○ role 的 defaults

定义该 role 使用的变量的默认值。这里设置的值一般都是无法直接运行的临时值或空值，实际的值在 group_vars 中描述，等到运行时再进行覆盖。这样一来只需看 defaults 就能掌握 role

所需的全部变量，使 role 的重复利用成为可能。

○ group_vars

针对 inventory 中定义的组进行设置。Ansible 会读取 group_vars 目录下的 YAML 格式文件。文件名对应组名。另外，文件名为 all，会对所有组套用变量。

○ host_vars

针对 inventory 中定义的主机进行设置。Ansible 会读取 host_vars 目录下的 YAML 格式文件。文件名对应组名。

11.2.2　文件结构

与 Ansible 关联的文件全都要在一个目录下统一管理。本章示例的目录结构如 LIST 11.3 所示。

LIST 11.3　ansible 脚本群的文件结构

```
+-- deployment/
    +-- group_vars/
        +-- all
        ...
    +-- host_vars/
    +-- roles/
         +-- environ
         +-- nginx
         +-- mysql
         ...
    +-- inventory/
        +-- production
        +-- dev
        ...
    +-- site.yml
    +-- ap.yml
    +-- lb.yml
    ...
```

这些文件也都要放在版本库中进行管理。管理方法可以有以下 2 种。

- 创建专用的版本库进行管理
- 与应用的源码放在同一个版本库中进行管理

◉ 在专用的版本库中管理

如果没必要或者不希望应用的开发与环境搭建同步，可以把源码与 Ansible 的文件群放在不同版本库中进行管理。

Ansible 关联文件的数量通常很大，如果不想让源码的版本库太复杂，同样可以用这个方法。

◉ **与应用的源码放在同一个版本库中进行管理**

在应用的版本库中专门创建一个用于管理这些文件的目录。这样做的好处是能在开发过程中让应用开发与环境搭建内容的变更对应起来。

我们大部分项目都采用了这种方法。没有特殊要求的情况下文件路径以 myproject/deployment/ 为基准。

11.2.3　执行 Ansible

创建好的脚本用 `ansible-playbook` 命令执行。对象环境的选择是通过切换 inventory 来实现的，因此要用 -i 选项明确指定 inventory。

```
$ ansible-playbook -i inventory/production.ini site.yml
```

在本章的结构中，会频繁用到指定了 tag 的执行操作。-t 选项可以在执行时指定任意标签。

```
$ ansible-playbook -i inventory/production -t deploy site.yml
```

11.2.4　与最初确定的结构相对应

11.1 节确定的搭建内容可以与 Ansible 的各概念对应起来。图 11.5 与图 11.2 相对应。

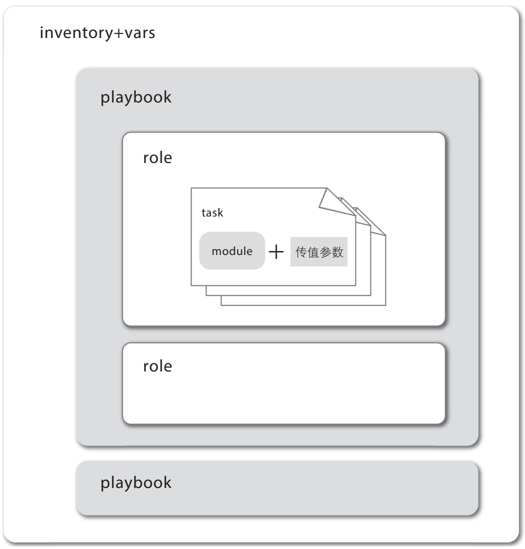


图 11.5　Ansible 的概念与服务器搭建内容的对应

- 环境 = inventory + vars

  针对各个环境（正式环境、开发环境、过渡环境等）编写inventory文件。另外，Ansible会自动读取与inventory的描述内容相对应的vars。
- 服务器组 = inventory 的段

  前面定义的服务器组在inventory中是以段形式定义的，各段中列举了主机。
- 各服务器组的搭建流程 = playbook

  给各个组分别创建lb.yml、ap.yml、db.yml、gateway.yml文件，将组设置成hosts。另外，创建一个include所有playbook的playbook，方便一次性搭建所有环境。这个playbook一般命名为site.yml。
- 针对各个中间件的搭建流程 = role

  以中间件为单位创建role，各服务器组用对应其所需role的playbook统一管理。
- 其他环境设置 = role

  不依赖于特定中间件的环境设置（创建用户等）也以role形式描述。

11.2.5　将各步骤 Ansible 化

◉ playbook/role 的设计

着手细节步骤之前，要先探讨搭建流程的各个步骤应该整合到怎样的 role/playbook 之中。我们对前面学习的搭建操作流程进行如下分类，然后分割到各个 role 中。

① 应该套用到所有服务器中的操作

除了本章讲到的创建用户之外，比较常见的还有 ntp 和时区设置。

我们将这些处理整合到名为 environ 的 role 中，对所有服务器组进行套用。

② 特定服务器组使用的中间件的设置

为每个中间件编写一个 role，以达到分割的目的。

③ 部署

包含到对象中间件的 role 里。设置 tag 以保证能单独执行部署操作。然后将分割好的 role 组合成 playbook。以下是正式环境的例子。

```
- hosts: app-servers
  roles:
    - environ
    - python
    - repository
    - django
```

```
    - appserver

- hosts: db-servers
  roles:
    - environ
    - mysql

- hosts: load-balancers
  roles:
    - environ
    - nginx
```

大致的设计完成后就可以开始实际编写所需操作，边测试边调整了。

◉ 用户设置

用户的创建和设置可以用 user 模块完成。

```
tasks:
  - user: name=mainte
  - user: name=www
```

借助 file 模块向 sudoers.d 目录下放置文件。本例的文件内容比较简单，因此可以通过 content 传值参数直接描述脚本内容来进行设置。

```
tasks:
  - file:
      dest: /etc/sudoers.d/mainte
      content: "%mainte ALL=(ALL) NOPASSWD:ALL"
```

公开密钥的设置用 authorized_keys 模块。传值参数处需要些公开密钥的字符串，由于字符串太长，我们用 group_vars/all 文件的变量代替。

```
mainte_pubkey: AAAA1234512345...
```

```
tasks:
  - authorized_keys:
      user: mainte
      key: "{{mainte_pubkey}}"
```

密钥文件在版本库中管理，通过 file 模块放置。这里不要忘记修改权限。

```
tasks:
  - file:
      src: mainte_seckey
```

```
      dest: /home/mainte/.ssh/id_rsa
      mode: 0600
```

本例的 src 只指定了文件名，它在这里其实是相对路径，该相对路径是基于下面两个中的一个。

- playbook 的目录
- 包含该 task 的 role 的 files 目录

◉ deb 程序包的安装

安装 deb 程序包时使用 apt 模块。指定版本的方式与 apt-get 相同。

```
- apt:
    name: mysql-server-5.5=5.5.40-0ubuntu0.14.04.1
```

用 `with_items` 可以将多个程序包的安装流程描述在一个 task 中。

```
- apt:
    name: "{{ item }}"
  with_items:
    - mysql-server-5.5=5.5.40-0ubuntu0.14.04.1
    - nginx
```

◉ 通过 pip 安装程序包

Ansible 有兼容 pip 的 pip 模块，使用前需要先安装 pip。

```
- name: install pip
  shell: curl -L https://bootstrap.pypa.io/get-pip.py | python creates=/usr/
local/bin/pip
```

使用 shell 模块时，如果传值参数 creates 处指定的文件已经存在，该步骤将被强制跳过。本例是通过检查是否存在 /user/local/bin/pip 来判断是否需要执行该步骤的。

由于这些模块都可以自由执行命令，因此幂等性需要我们自己来验证。只要给传值参数 creates 指定了文件，当被指定的文件存在时，当前步骤就会被强制跳过。本例是通过检查 /user/local/bin/pip 是否存在来判断是否需要执行该步骤。

安装完成后 pip 模块就能用了。

```
- name install packages
  pip: name={{item}}
  with_items:
    - virtualenv
```

我们在第 1 章中也了解到，应用所需的程序包要安装在 virtualenv 环境中。因此我们把搭建 virtualenv 环境的步骤也写了进来。

```
- name: create virtualenv
  sudo_user: www
  command: virtualenv venv chdir=/var/www/ creates=venv/bin/activate

- name: install
  sudo_user: www
  pip: requirements=/var/www/myproject/requirements.txt virtualenv=/var/www/venv
```

pip 模块会向 virtualenv 参数处指定的环境安装程序包，所以我们在这里指定虚拟环境的路径。另外，只把 name 参数替换为 requirements 参数并指定 requirements.txt 的路径，就可以使用 requirements.txt 了。

◉ 中间件设置的反映

放置、更新设置文件要用 template 模块。

之前我们在设置文件中使用了一些尚未定义的变量，现在在 group_vars 和 role 的 defaults/main.yml 中定义它们。比如对 nginx.conf 中使用的变量要作如下定义（LIST 11.4）。

LIST 11.4　group_vars/all

```
DOMAIN: myproject.example.com
NGINX_UPSTREAMS:
  - 192.168.0.3
  - 192.168.0.4

SSL:
  CERTIFICATE: myproject.crt
  KEY: myproject.key
```

变量准备完毕后开始写放置配置文件的 task。涉及多个文件时推荐使用 with_items。

```
tasks:
  - template:
      src: "{{ item }}"
      dest: /etc/nginx/"{{ item }}"
    with_items:
      - conf.d/gunicorn.conf
```

template 模块的文件路径引用的是包含该 task 的 role 的 templates 目录。保持 templates 目录的相对路径与文件放置目标目录的相对路径一致能够简化描述。

◉ 部署

部署的 task 也写在各个 role 的 tasks 中。描述时要给该 task 设置 tag。设置 tag 能让我们单独拿出与部署相关的 task 来执行。

按照本章的结构，我们定义下述 tag。

- configure
  设置文件的更新
- update
  源码的更新
- reload
  重新加载中间件设置
- deploy
  执行update、configure、reload

◉ 设置的更新

给在“中间件设置的反映”部分编写的 template 模块的 task 添加 configure 和 deploy 标签。

```
- template:
    src: "{{ item }}"
    dest: /etc/nginx/"{{ item }}"
  with_items:
    - conf.d/gunicorn.conf
  tags:
    - configure
    - deploy
```

◉ 中间件的重启

经由 sysvinit 或 upstart 启动的中间件可以用 service 模块重启或重新加载。

```
- service:
    name: nginx
    state: reloaded
  tags:
    - reload
    - deploy
```

如果使其与 Ansible 的 Notify 机制相结合，则可以规定仅在设置文件被修改时自动重新加载中间件。

通过 Notify 调用的 task 在 role 的 handlers/main.yml 中描述。name 会被视为标识符，所以要指定一个在整个项目中独一无二的名字。

LIST 11.5　roles/nginx/handlers/main.yml

```
- name: reload nginx
  service:
    name: nginx
    state: reloaded
```

接下来在调用方的 notify 参数处指定通过 LIST 11.5 设置的 name。这个 task 的返回值 changed 为 True 时（对 template 模块而言是文件内容被更新时）将执行 notify 指定的 task（LIST 11.6）。

LIST 11.6　roles/nginx/tasks/main.yml

```
...
- name: configure nginx
  template:
    src: "{{ item }}"
    dest: /etc/nginx/"{{ item }}"
  with_items:
    - conf.d/gunicorn.conf
  tags:
    - configure
    - deploy
  notify:
    - reload nginx
```

使用 Notify 可以让我们不必分神去注意重新加载设置的问题，但会使单独执行重新加载或重启，以及只修改设置不重启等类似操作变得难以实现。因此需要根据项目的实际情况选择合适的方法。

◉ 源码的更新

通过 Mercurial 放置源码的操作可以用 hg 模块来完成，但需要事前完成安装 Mercurial、创建目录、设置权限等准备工作。

把这一系列工作整合到名为 repository 的 role 中，让需要放置源码的服务器组能够使用该 role。

```
- pip:
    name: mercurial

- file:
    state: directory
    dest: /var/www/
```

```
    owner: www
    mode: 755

- hg:
    repo: ssh://user@myrepository.com/myproject
    dest: /var/www/myproject
    revision: "{{ revision }}"
  tags:
    - update
    - deploy
```

为了能够更新成任意版本而不仅限于 default，这里将 hg 模块的传值参数 revision 设为变量。

根据需要还可以将放置源码的目标项目名、版本库 URL 也设为变量，方便替换和引用，从而灵活应对多种需求。

11.2.6　整理 Ansible 的执行环境

本例中的环境如 11.1.1 节所述，除 gateway 服务器以外全都无法从外部进行 ssh，因此我们无法从本地环境操作所有服务器。另外，如果给所有相关人员的本地环境中都搭建 Ansible 的执行环境，那么每次调换或新增负责人时都要搭建一次环境。

这种时候，最好的解决办法就是在一个项目全体成员共享的、可连接环境中所有服务器并且与应用运行没有直接关系的 gateway 服务器上搭建 Ansible 的执行环境。

虽然我们也希望通过 Ansible 完成 gateway 服务器的整理，但这又涉及到用 Ansible 整理 Ansible 执行环境的自举问题。

这里我们从手头环境执行 Ansilbe，以解决这一问题。但有一点要注意，执行这部分操作时 gateway 服务器上还不存在 mainte 用户。虽然我们设想用 mainte 进行维护工作，但构建环境时无法使用该用户。所以要为 root 或 ubuntu 用户创建一份临时的 inventory 文件（LIST 11.7 ～ LIST 11.9）。

LIST 11.7　gateway 搭建时所需的 ini

```
[gateway]
gateway.example.com ansible_ssh_user=ubuntu
```

LIST 11.8　gateway.yml

```
- hosts: gateway
  sudo: yes
  roles:
    - environ
```

```
    - python
    - ansible
```

LIST 11.9　roles/ansible/tasks/main.yml

```
- name: install Ansible
  pip: name=ansible
```

11.3　小结

本章讲解了探讨环境搭建流程的思路以及自动化搭建环境的方法。

我们往往觉得环境的运行没有什么技术含量，而且很难看到付诸其中的努力。但是，应用能持续平稳地运行，毫无疑问是高效环境搭建以及维护的功劳。

从长远角度看，搭建环境的自动化对提高搭建环境及维护的效率有十分明显的效果。另外，搭建流程自动化使得项目成员能够自由地搭建个人开发环境，能大幅提高整个项目的生产能力。

因此推荐各位搭建环境时多注意稳定和高效两个方面。另外，为了让项目稳定高效运转，建议导入环境搭建的自动化。

专栏　自动化的“度”在哪里

当我们为实现自动化而写搭建流程的详细列表时，会发现搭建所需的步骤比我们想象中要多得多。其中有些内容自动化起来很麻烦，有些内容又很难自动化。于是这些步骤要自动化到一个什么“程度”便成了重要的研究课题。

刚开始导入自动化时，看到服务器自己搭建环境，人们往往会得意忘形，希望把所有步骤全都自动化。然而我们认为这样做是错的。Ansible 的 Playbook 虽然能极简洁地描述搭建步骤，但量堆积到一定数量同样会使可读性变差。shell 模块确实可以强行自动化一些繁杂的步骤，不过日后读和改的时候肯定会遇到麻烦。

我们认为，简化流程是流程自动化工作的一部分。大部分自动化工具都把构建时经常遇到的操作进行了简化，使得我们能轻松执行这些操作，Ansible 更是将这些机制以标准模块的形式提供给了我们。难以用这些标准模块实现的操作可以认为是冗余的或者是错的，应该考虑删除或者改良。有些时候，如果能简化流程或结构，考虑改变架构也是值得的。

通过自动化实现繁琐的搭建流程远不如找一个能简单完成环境搭建的方法来得有价值。因此不要想着用自动化去掩盖复杂的操作，而是要以自动化为契机着手改善流程。

专栏　巧用备份

搭建环境实现自动化后，每次向组中添加服务器都要整体从零搭建环境。但是，一遍遍重新搭建相同结构的服务器是一件非常浪费时间的事。如果我们采用的基础设备可以使用备份或服务器镜像，那么直接复制现有的服务器备份要远比重新搭建环境轻松且安全。

另外，添加新的服务器组时，由于全新服务器中不存在维护用户，所以每次都要面对自举问题。如果能从已有维护用户的状态开始搭建，那么整个过程将轻松不少。

出于以上原因，我们在搭建时会按照下述方针进行备份，缩短搭建时间。

- 在执行了 environ role 的状态下做一次备份
- 基于上述备份给每组里的每一台服务器搭建环境并备份，有多台服务器组成的服务器组基于该备份搭建环境。

虽然看服务器自动搭建环境是一种享受，但多余操作还是应该尽量减少。

第 12 章　应用的性能改善

使用 Web 应用时，随着访问量的增加，我们会遇到响应延迟、请求失败无法正常提供服务的情况。

本章我们先学习上述问题的应对方法以及 Web 应用 / 服务器的性能评估，然后阶段性地导入 gunicorn、nginx，并在各阶段进行性能评估，观察性能改善的情况。

本章最终将形成如图 12.1 所示的服务器结构。gunicorn 和 nginx 的相关内容会在导入时详细说明，这里不作深入了解。

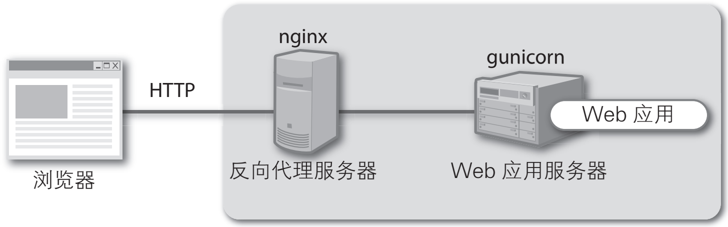


图 12.1　nginx 与 gunicorn 的协作

> NOTE
>
> 所谓反向代理（Reverse Proxy），是指负责接收随机多个客户端发来的请求的服务器，其目的在于降低特定服务器处理请求的负担或向指定服务器群分配访问请求。它的运作方式与通常的代理服务器正好相反，因此称为反向代理。

12.1　Web 应用的性能

Web 应用负荷过重时会产生哪些问题？面对负荷过重应采取什么对策？选择对策时又应该进行哪些考量呢？接下来我们将了解一下这些问题。

12.1.1　Web 应用面对大量集中请求时会产生哪些问题

当应用服务器接收到的请求增多时，如果一个请求尚未处理完又接到了下一个请求（即两个请求几乎同时到达），那么后一个请求将被加入队列，等待前一个请求处理完毕。即便是可以并行处理多个请求的服务器，一旦到了并行处理数的极限，后到的请求也会被加入队列，等待

处理的线程将越来越多。

这种状态会使应用服务器进入高负荷状态，出现 CPU 负担加重、内存空间不足等问题。

高负荷状态下应用可能无法正常运行，或者性能出现明显下降。以第 2 章中编写的 Web 应用为例，这个应用在 Python 的 SimpleHTTPServer 模块的 Web 应用服务器上运行，然而这个服务器是单进程单线程的，无法同时处理多个请求。因此请求集中到达时等待处理的请求会增多，应用性能会下降。

另外，等待队列也是有上限的，具体上限与硬件本身的性能有关。一旦等待数达到上限，新的请求将被视为无法处理的请求，表现为切断连接，请求失败。另外，队列达到上限有时会造成服务器的程序异常停止，导致无法连接服务器。

这些问题在客户端会表现为无法连接、反应慢、频繁报错等。该状态会影响用户对应用的正常使用。

12.1.2　针对高负荷的对策

解决高负荷状态需要哪些对策呢？

高负荷其实可以细分为很多种，不同种的负荷解决方法也不同。比如应用服务器的 CPU 处理能力不足了，结果我们换了一块性能更好的硬盘，这显然无法解决问题。

因此要明确问题所在，选择合适的对策。下面是几种典型问题及其对策。

问题	解决方法
CPU 占用率高	增加处理的进程和线程数。更换成性能更好的 CPU
硬盘 I/O 负荷高	优化读取、写入数据的方法。更换成性能更好的硬盘。
内存不足	释放被无用程序占用、分配的内存。增加物理内存。

除此之外还有许多解决方法，不过本章将以上述 3 点为中心进行说明。另外，为提高改善性能的效率，需要遵循以下原则。

- 从预期效果最大的方法开始尝试
- 从所需资金、步骤、时间最少的方法开始尝试
- 实施对策前后各评估一次性能

第一条“从预期效果最大的方法开始尝试”是因为如果先采用了效果较小的方法，日后再采用效果较大的方法时，前一个方法的效果会被覆盖掉，这就使前一次的工作成了无用功。

第二条“从所需资金、步骤、时间最少的方法开始尝试”指优先选择性价比高的方法，把成本高成效低的方法摆到次要位置。

第三条“实施对策前后各评估一次性能”是为了掌握该对策的实际效果。评估结果能明确告诉我们成本换来了多少效果。

12.2　评估留言板应用的性能

本节，我们将对第 2 章中开发的应用进行性能评估，然后学习如何让该应用在 nginx 和 gunicorn 上运行。

12.2.1　什么是应用的性能

我们总是会说“应用的性能好 / 不好”，但好与不好究竟指的是什么呢？下面是几个性能好的例子。

- 应用的处理能力强
- 内存占用量小
- 页面显示速度快
- 通信响应速度快

这些评价对象都不相同，评价中使用的词语也是强、小、快，各不一样。可见，应用的性能有多个指标。所以在谈论性能时，必须首先确定是哪方面性能，不然理解上就会出现偏差。

在 Web 应用的各项性能指标中，本章将重点研究从发送 HTTP 请求到返回响应的这段时间（即响应时间）。响应时间越短，浏览器切换页面的速度就越快，用户就会觉得应用越流畅。另外，我们在评估性能时将使用 ApacheBench（ab 命令）。这是一款基准测试工具，能简单地检测应用的响应时间和请求成功率。

12.2.2　安装 ApacheBench

ApacheBench 是 Web 服务器 Apache 附带的工具之一。通过 apt 进行安装时，要安装 apache-utils（LIST 12.1）。

LIST 12.1　通过 apt 进行安装

```
$ sudo apt-get install apache2-utils
```

安装完后就能使用 ab 命令了。本章使用的是 ApacheBench 的 2.3 版本。

12.2.3　用 ApachBench 评估性能

这里运行第 2 章中开发的应用，用 ab 命令检测响应时间。这里以留言板首页的响应速度为检测对象。我们希望在已提交数据的状态下进行检测，因此事先输入了 50 条数据。另外，应用的运行环境如下。

OS	Ubuntu 14.04（64bit 版）
CPU	Intel Core 2 Duo 2.2GHz（双核）
内存	512MB

检测前先启动应用服务器（LIST 12.2）。

LIST 12.2　启动应用

```
$ python guestbook.py
 * Running on http://127.0.0.1:8000/
 * Restarting with reloader
```

使用 screen 命令的情况下会弹出新窗口，在新窗口中执行 ab 命令。不使用 screen 命令时则需要按 Ctrl+Z 键，将运行中的应用移至后台，然后再执行 ab 命令。

应用可以通过 IP 地址 127.0.0.1、端口号 8000 访问，所以 ab 命令要指定这个 URL（LIST 12.3）。-n 选项可以指定请求次数，-c 选项可以指定连接数。这里我们把总计 1000 次请求分 100 个连接并行发送。

LIST 12.3　执行 ab 命令

```
$ ab -n 1000 -c 100 http://127.0.0.1:8000/
```

检测结束后会输出如 LIST 12.4 所示的报告。

LIST 12.4　ab 命令的结果

```
Server Software:        Werkzeug/0.9.6
Server Hostname:        127.0.0.1
Server Port:            8000

Document Path:          /
Document Length:        7398 bytes

Concurrency Level:      100
Time taken for tests:   4.911 seconds
Complete requests:      1000
Failed requests:        0
```

```
Write errors:           0
Total transferred:      7554000 bytes
HTML transferred:       7398000 bytes
Requests per second:    203.61 [#/sec] (mean)
Time per request:       491.146 [ms] (mean)
Time per request:       4.911 [ms] (mean, across all concurrent requests)
Transfer rate:          1501.99 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.1      0       5
Processing:    19  466  82.5    489     497
Waiting:       19  466  82.5    489     497
Total:         24  467  81.5    489     497

Percentage of the requests served within a certain time (ms)
  50%    489
  66%    490
  75%    491
  80%    491
  90%    492
  95%    496
  98%    497
  99%    497
 100%    497 (longest request)
```

接下来了解一下检测结果中需要注意的地方。

◉ Complete requests

Complete requests 部分显示了请求成功与失败的情况。通过上面的例子，我们可以看到，1000 个请求全部成功。如果请求没有全部成功，表示可能超出了 Web 服务器的处理能力。

```
Complete requests:      1000
Failed requests:        0
```

◉ Requests per second

1 秒处理的请求数。下述结果表示每秒大约处理 203 个请求。

```
Requests per second:    203.61 [#/sec] (mean)
```

◉ Connection Times

Connection Times 部分可以查看处理 1 个请求所需的连接时间（Connect）、处理时间（Processing）、等待时间（Waiting），单位精确到毫秒。各列元素分别代表最小值（min）、平均值（mean）、标准差（[+/-sd]）、中间值（median）、最大值（max）。如果某个时间存在明显的不平均现象，证明应用的某个部分有可能存在瓶颈。

以下述执行结果为例，我们是在本地环境中连接本地的 Web 应用，所以连接时间非常短。另外，处理时间和等待时间都没有明显的不平均现象，证明应用中不存在瓶颈。

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.1      0       5
Processing:    19  466  82.5    489     497
Waiting:       19  466  82.5    489     497
Total:         24  467  81.5    489     497
```

我们的这个留言板应用直接使用了 Flask 内置的 HTTP 服务器。该内置 HTTP 服务器由 Python 标准模块的 HTTPServer 类扩展而来。这个服务器并不算特别慢，但如果导入更高速的 Python 专用应用服务器，将能明显改善响应性能。

接下来我们将导入 gunicorn，看看这个 Python 的面向 Web 应用的 HTTP 服务器是如何改善响应性能的。

NOTE

本章用 ApacheBench 评估了应用的性能，但要注意，这与 Web 应用实际运作时的情况不完全相同。

ApacheBench 的评估对象是单一 URL，但一般的 Web 应用除 HTML 以外还需要对图片、CSS、JavaScript 文件发送请求。要获取某 URL 页面中的所有元素，有时需要几十个请求。因此各位要记住，ab 命令检测出的响应性能与用户体感的响应时间之间是存在差异的。

12.3　gunicorn 简介

gunicorn 是面向 Python WSGI 应用的 HTTP 服务器。它在 Linux/Unix 上运行，优点是轻量级且速度快。gunicorn 通过一个控制进程和多个工作进程来执行应用程序。由于它是一个 Python 模块，所以可以变更工作进程的类以及扩展服务器本身。

12.3.1　安装 gunicorn

gunicorn 可以用 pip 命令安装（LIST 12.5）。

LIST 12.5　用 pip 命令安装

```
$ pip install gunicorn
```

通过上述方法安装 gunicorn 后，就可以使用 `gunicorn` 命令了。

12.3.2　在 gunicorn 上运行应用

首先设置一个工作进程，运行应用（LIST 12.6）。

LIST 12.6　用 gunicorn 命令运行留言板应用

```
$ gunicorn -w 1 -b 127.0.0.1:8000 guestbook
```

我们在这个状态下再执行一次 `ab` 命令，检测结果如下。

```
Server Software:        gunicorn/19.1.1
Server Hostname:        127.0.0.1
Server Port:            8000

Document Path:          /
Document Length:        7398 bytes

Concurrency Level:      100
Time taken for tests:   4.469 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      7560000 bytes
HTML transferred:       7398000 bytes
Requests per second:    223.75 [#/sec] (mean)
Time per request:       446.937 [ms] (mean)
Time per request:       4.469 [ms] (mean, across all concurrent requests)
Transfer rate:          1651.87 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   2.2      0       8
Processing:    31  424  71.7    444     462
Waiting:       14  424  71.8    443     462
```

```
Total:          37  425  69.8    444     469

Percentage of the requests served within a certain time (ms)
  50%    444
  66%    444
  75%    444
  80%    444
  90%    445
  95%    445
  98%    446
  99%    446
 100%    469 (longest request)
```

可以看到与之前用 Flask 内置服务器时的结果差别不大。

由于当前运行应用的虚拟机的 CPU 数设置为 2，因此单独一个工作进程无法充分利用 CPU 资源。所以进程数应至少设置为 2。

于是我们将 gunicorn 的工作进程设置为 2，再次执行 ab 命令进行检测。工作进程数用 -w 选项指定（LIST 12.7）。

LIST 12.7　用 gunicorn 命令运行应用（工作进程数为 2）

```
$ gunicorn -w 2 -b 127.0.0.1:8000 guestbook
```

检测结果如下。

```
Server Software:        gunicorn/19.1.1
Server Hostname:        127.0.0.1
Server Port:            8000

Document Path:          /
Document Length:        7398 bytes

Concurrency Level:      100
Time taken for tests:   2.376 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      7560000 bytes
HTML transferred:       7398000 bytes
Requests per second:    420.91 [#/sec] (mean)
Time per request:       237.582 [ms] (mean)
Time per request:       2.376 [ms] (mean, across all concurrent requests)
Transfer rate:          3107.48 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.6      0       7
Processing:    18  225  37.3    235     248
Waiting:       17  225  37.3    235     247
Total:         25  226  35.8    235     252

Percentage of the requests served within a certain time (ms)
  50%    235
  66%    236
  75%    236
  80%    236
  90%    236
  95%    237
  98%    238
  99%    238
 100%    252 (longest request)
```

平均响应时间缩短了 40%，可见速度快了许多。接下来我们再看看用作 Web 服务器和代理服务器的 nginx。

12.4　nginx 简介

nginx（Engine X）[1] 是 Nginx Inc. 开发的 Web 服务器，拥有 HTTP 服务器、代理服务器等功能，轻量级且速度快。

近年来 nginx 成长迅速，在 Web 服务器界占有的份额仅次于 Apache、IIS（Internet Information Services），居世界第三（约 14%）。一些较大规模的 Web 服务也采用了 nginx，比如 WordPress.com[2]、GitHub[3]、Instagram[4] 等。

12.4.1　安装 nginx

nginx 的安装可以直接从源码构建，不过由于 Ubuntu 的版本库中有相关程序包，所以我们

① http://nginx.org/
② https://wordpress.com/
③ https://github.com/
④ http://instagram.com

用 apt-get 命令进行安装（LIST 12.8）。

LIST 12.8 安装 nginx

```
$ sudo apt-get install nginx
```

本书使用了 nginx 的 1.4.6 版本（LIST 12.9）。

LIST 12.9 查看版本

```
$ nginx -v
nginx version: nginx/1.4.6 (Ubuntu)
```

启动、停止等操作通过 service 命令实现（LIST 12.10 ～ LIST 12.13）。

LIST 12.10 启动 nginx

```
$ sudo service nginx start
```

LIST 12.11 停止 nginx

```
$ sudo service nginx stop
```

LIST 12.12 重启 nginx

```
$ sudo service nginx restart
```

LIST 12.13 重载 nginx

```
$ sudo service nginx reload
```

测试设置文件是否有错时，执行配置测试（ConfigTest）（LIST 12.14）。

LIST 12.14 nginx 的配置测试

```
$ sudo nginx -t
```

12.4.2 检测 nginx 的性能

前面我们检测了留言板应用的性能，现在我们看看 Web 服务器 nginx 本身的性能如何。

由于要运行默认站点，所以这里我们先用管理员权限编辑设置文件 /etc/nginx/sites-available/default（LIST 12.15）。如果 Apache 等已安装的软件已经占用了端口 80，则需要将 listen 后的端口号修改成其他数值。

LIST 12.15 /etc/nginx/sites-available/default

```
server {
    listen    80;  # 使用端口 80
    server_name  localhost;  # 主机名为 localhost
```

```
    # 访问日志的文件路径
    access_log  /var/log/nginx/localhost.access.log;

    location / {  # 域名后的路径为 / 时
        root   /var/www;  # 根目录
        index  index.html index.htm;  # 索引文件的文件名
    }
}
```

这里不存在已设置好的根目录 /var/www 时，需要手动创建（LIST 12.16）。

LIST 12.16　创建 /var/www 目录

```
$ sudo mkdir /var/www
$ sudo chown www-data:www-data /var/www
```

用管理员权限生成用于默认站点的目录（LIST 12.17）。

LIST 12.17　/var/www/index.html

```
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title> 测试页面 </title>
  </head>
  <body>
    <h1> 测试页面 </h1>
  </body>
</html>
```

另外，有时候我们会遇到工作进程数默认为 1 的情况，所以要根据 CPU 数（核心数）修改 /etc/nginx/nginx.conf 的 worker_processes。这里我们设置为 2（LIST 12.18）。

LIST 12.18　/etc/nginx/nginx.conf

```
user www-data;
worker_processes 2;

# 下略
```

编辑完成后执行配置测试，确认没有问题后重载或重启 nginx（LIST 12.19）。

LIST 12.19　nginx 的配置测试与重载

```
$ sudo nginx -t
the configuration file /etc/nginx/nginx.conf syntax is ok
configuration file /etc/nginx/nginx.conf test is successful
```

```
$ sudo service nginx reload
* Reloading nginx configuration nginx                            [ OK ]
```

接下来用 ApacheBench 评估性能（LIST 12.20）。

LIST 12.20　执行 ab 命令

```
$ ab -n 1000 -c 100 http://127.0.0.1/
```

评估结果如下。

```
Server Software:        nginx/1.4.6
Server Hostname:        127.0.0.1
Server Port:            80

Document Path:          /
Document Length:        194 bytes

Concurrency Level:      100
Time taken for tests:   0.214 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      404000 bytes
HTML transferred:       194000 bytes
Requests per second:    4662.92 [#/sec] (mean)
Time per request:       21.446 [ms] (mean)
Time per request:       0.214 [ms] (mean, across all concurrent requests)
Transfer rate:          1839.67 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    9   2.7     10      11
Processing:     1   12   3.8     11      26
Waiting:        1   10   4.1      9      25
Total:         11   21   2.9     21      29

Percentage of the requests served within a certain time (ms)
  50%     21
  66%     21
  75%     21
  80%     21
  90%     23
  95%     26
  98%     28
```

```
  99%     29
 100%     29 (longest request)
```

与 gunicorn 上运行的留言板相比，响应速度快出了一个层级。

12.5　在 nginx 和 gunicorn 上运行应用

现在在 gunicorn 上运行留言板应用，然后通过 nginx 进行反向代理，返回响应。通过反向代理响应可以将留言板应用的服务器横向扩展成多个。这里我们不进行横向扩展，只评估单一服务器的应用性能。

12.5.1　gunicorn 的设置

gunicorn 可以指定 -D 选项将进程转为守护进程，即守护进程化（Daemonize）（LIST 12.21）。

LIST 12.21　通过 gunicorn 命令将留言板应用转为守护进程并执行

```
$ gunicorn -w 2 -b 127.0.0.1:8000 -D guestbook
```

NOTE

守护进程化是指以 Unix 守护进程的形式运行程序。Unix 守护进程运行在后台，并且启动之后不需要用户直接控制。

以 gunicorn 为例，给 gunicorn 命令指定 -D 选项后，gunicorn 便脱离了用户的控制，会转为持续等待请求的后台进程。

12.5.2　nginx 的设置

编辑 nginx 默认站点的设置文件，以反向代理的形式连接在 gunicorn 上运行的应用（LIST 12.22）。

LIST 12.22　/etc/nginx/sites-available/default

```
# 名为 guestbook 的 upstream（上游服务器）的设置
upstream guestbook {
    server 127.0.0.1:8000;
}

server {
```

```
    listen   80;  # 使用端口 80
    server_name  localhost;  # 主机名为 localhost
    # 访问日志的文件路径
    access_log  /var/log/nginx/localhost.access.log;

    location / {  # 域名后的路径为 / 时
        # 反向代理到名为 guestbook 的 upstream
        proxy_pass http://guestbook;
    }
}
```

upstream 指令里可以设置多个服务器。在一台应用服务器无法满足处理需求时，可以在 upstream 里设置多台应用服务器来分担负荷。

设置文件编辑完成后要记得进行配置测试及重载。

12.5.3　评估 nginx+gunicorn 的性能

接下来用 ApacheBench 评估性能（LIST 12.23）。

LIST 12.23　执行 ab 命令

```
$ ab -n 1000 -c 100 http://127.0.0.1/
```

评估结果如下。

```
Server Software:        nginx/1.4.6
Server Hostname:        127.0.0.1
Server Port:            80

Document Path:          /
Document Length:        7398 bytes

Concurrency Level:      100
Time taken for tests:   2.655 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      7556000 bytes
HTML transferred:       7398000 bytes
Requests per second:    376.61 [#/sec] (mean)
Time per request:       265.529 [ms] (mean)
Time per request:       2.655 [ms] (mean, across all concurrent requests)
Transfer rate:          2778.95 [Kbytes/sec] received
```

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   3.6      0      14
Processing:    22  252  42.2    264     273
Waiting:       22  252  42.2    264     273
Total:         27  253  39.8    264     286

Percentage of the requests served within a certain time (ms)
  50%    264
  66%    265
  75%    265
  80%    265
  90%    266
  95%    266
  98%    267
  99%    268
 100%    286 (longest request)
```

与单独由 gunicorn 构成的服务器相比，响应时间略有延长，但考虑到可用多台服务器构成服务器组，这点损失还是可以接受的。

12.5.4　性能比较

最后我们来比较一下改善前和改善后的性能评估结果。评估结果如 LIST 12.24 和 LIST 12.25 所示。

LIST 12.24　改善前的评估结果（精选）

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.1      0       5
Processing:    19  466  82.5    489     497
Waiting:       19  466  82.5    489     497
Total:         24  467  81.5    489     497
```

LIST 12.25　改善后的评估结果（精选）

```
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   3.6      0      14
Processing:    22  252  42.2    264     273
Waiting:       22  252  42.2    264     273
Total:         27  253  39.8    264     286
```

从 Flask 的开发服务器换到 nginx 和 gunicorn 组成的服务器之后，Total 的时间比改善前缩短了近一半。响应时间缩短了，因此我们可以认为有改善效果。至此改善工作结束。

NOTE

若想在上述基础上进一步缩短响应时间，可以考虑下述对策。出于篇幅原因，这里就不进行详细说明了。

- 数据库方面将 shelve 换成 MySQL 等 REBMS（加快数据库访问速度）
- 在别的机器上运行应用，增加机器数（增加可同时处理的请求数）
- 使用 memcached 等缓存服务器（减少访问数据库的次数）

12.6　小结

本章首先讲了对 Web 应用集中访问时会产生的服务器负担加重、应用运行不稳定等问题。

接下来介绍了该类问题的解决方法以及性能的概念，并用 ApacheBench 对第 2 章中开发的 Web 应用进行了性能评估。另外，还讲解了如何在 gunicorn 这一应用服务器上运行 Web 应用，以及如何设置 nginx 用作反向代理。

改善性能的目的是让已有系统的性能最优化。然而，在我们花大块时间一次次寻求改善时，会发现新解决方案的效果总是越来越小。因此，改善性能之前要先定一个性能的目标值，确定要花多少时间来做这件事。着手改善前切记要先对现有系统进行评估，通过比较确定目标。选一个合适的目标，用最少的时间和资源完成最优化工作，这才是我们追求的高效的性能改善。

第 4 部分

加速开发的技巧

第 4 部分的内容与团队开发、正式环境等话题相对独立。这里将介绍完成集成测试之后的测试思路、在 Django 上进行开发时需要注意的信息、方便好用的 Python 库等内容。

第 13 章　让测试为我们服务

不知各位在开发过程中有没有遇到过下述情况。

- 没有明确的目标，只能摸索着进行开发，最后成品与原定的需求不符，导致整个推翻重做
- 程序严格按照设计书实现，但在集成测试阶段发现其与周边功能结合不到一起，导致故障频出

本章将把测试的观点引入开发的各个过程之中，为解决或规避开发中出现的上述问题提供解决方案。我们希望本章内容能让各位从开发初级阶段开始就知道需要兼顾测试，明白自己需要考虑什么，需要亲眼确认什么，为各位的整个开发过程带来帮助。

NOTE

本书的中心内容是编程技法，因此不对各个测试所用的手法进行具体讲解。测试手法的相关内容有许多优秀的专业书籍可供参考，有兴趣的读者不妨一试。

·《软件测试的艺术》(机械工业出版社，2012 年)

·《软件测试 PRESS 合集》[①](技术评论社，2011 年)

13.1　认识现状：测试的客观环境

在各位所处的环境中，测试处于怎样一个地位呢？

在从确认运行状况到进行系统测试的过程中，各个阶段都花大把时间去测试吗？还是在编码结束后只简单确认一下运行状况就发布了呢？又或者是仔细编写测试代码，但却尽量控制手动测试呢？

细化到各个测试阶段来说，由于项目的性质、交付对象 / 搭档 / 经理的方针等因素的影响，很多时候我们不能选择最合适的工作流程，所以时间安排往往不尽如人意。另外，就算日程表已经制定好，某些时间安排也会因各种情况而缩短甚至删除。

所以很多时候我们心里明白测试的重要性，却实在挤不出时间给测试。对于需要赶工的项目而言更是如此。面对火烧眉毛的日程表，我们往往会失去方向，不知道该从哪里下手。

① 原书名为『ソフトウェア・テスト PRESS 総集編』，暂无中文版。——编者注

13.2　将测试导入开发各个阶段

项目开始之后，为了能尽可能高效地完成开发的各阶段，即便时间再紧，也要尽早将测试的观点引入项目之中。因为随着开发流程的推进，修改、变更都会越来越难。

这就像身处一个陌生的环境，只要在行动前先确认一下地图，了解当前地点和目的地的位置，之后就基本不会走偏。即便方向略有偏差，只要能在转第一个弯时及时发现，我们依然能很快返回出发点。也就是说，如果途中能每到一个路口都确认一下，就算其间走错过一条路，我们也能很快折返或者在下一个路口修正方向。但是，如果行动前不作任何确认，等瞎蒙乱撞地跑到了其他城市才意识到走错了路，此时再回过头来看地图就为时已晚了。到了这个地步，不但向前走找不到目的地，就连折返也很难返回出发点，进退两难。这在系统开发上也是同样道理，我们开始项目之前必须正确理解项目的前提条件和目的，尽量在初期阶段及早发现错误，修正方向。

总之，所谓测试的观点，就是给项目定义一个正确的状态，然后在开发过程中及时发现偏离该状态的元素并对其加以改善。将测试的观点从程序测试阶段扩展至需求定义和设计阶段，这能有效地帮助我们在开发初期修正错误。

接下来我们按照开发流程一步步进行了解。

13.2.1　文档的测试（审查）

文档类的测试一般称为审查（Review），它也是测试的一种。认真进行审查能防止因理解偏差而导致的返工或者不小心实现了多余内容等问题。因此即便没有充足时间去单独审查每个文档，也要在开始实现或测试设计之前简单地将文档重新审视一遍。

文档可分为很多种，比如下面这些。

【主要文档类型】

- 需求说明
- 需求定义
- 基本设计
- 详细设计
- ER 图
- 数据表定义
- 测试说明或测试设计等

文档大致可分为三大类，一是确定开发方向的文档（需求说明、需求定义等），二是指导具体

实现的文档（基本设计、详细设计等），三是规定数据使用方式的文档（ER 图、数据表定义等）。

下面是审查文档时经常发生的问题。

【问题】

- 文档没有更新，内容陈旧
- 未指明未确定的事项
- 描述含糊不清
- 存在多个相同标题的文档文件，搞不清哪个才是最新的

对于内容陈旧的文档，只能随发现随更新，然后定期重审，没有其他办法。而其他问题都是有改善余地的，下面我们来一个个解决。

◉ 未确定的事项标明“未确定”

文档是担保程序以及测试正确性的重要指标，因此其内容务必准确详实，不能有遗漏，也不能有多余内容。

拿到文档时，不要直接动手开工，先从头到尾通读一遍文档，把其中有疑问的地方和不清楚的地方记录下来。特别是在项目刚刚开始的阶段，某些需求和规定还很模糊。此时要把不明确的部分找出来，在对象文档里注明“未确定”。明确未确定的项目是对残留的待确定项目的一种可视化，让我们能直观地看到哪里还有待确定的项目。想消除文档的遗漏、缺失等问题，这种可视化必不可少。“不清楚还有哪些不清楚的事”的情况必须尽早解决。记录未确定项目时可以同时将确定该项目所需的条件写下来，便于重审文档时确认是否需要处理该项目。

另外，查找文档漏记了哪些内容其实是一个难度很高的工作，这里介绍两种行之有效的方法。

一种是对比上一阶段的文档，查看内容是否有遗漏。由于当前阶段文档全部源于前一阶段文档，所以二者内容出现不吻合时，可以认为文档某处存在遗漏。

另一种是将功能、页面、数据等方面有关联的文档放在一起，看它们之间是否存在矛盾。如果对比文档时出现内容不契合的情况，那么文档某处很可能存在问题。

【消除不明确的点】

- 存在未确定的事项时，在对应文档内标明“未确定”
- 标注“未确定”的同时记录确定所需的条件
- 查看前一阶段的文档，确认项目是否存在遗漏
- 检查功能、页面、数据等方面的关联，确认各部分关联正常

◉ 描述时尽量避免歧义

如果阅读文档时发现内容存在模糊不清的地方，证明我们对正确的内容的定义还不够明确。

- 在表达同一个事物时用了不同说法（用语不一致）
- “○○等”“○○左右”等需要清单化的部分表述含糊不清
- 数值增减的幅度、上下限不明（比如写 1 ～ 10，看的人不清楚是按 1、2、3…还是按 1.0、1.1、1.2…的规律变化）
- 能数值化、符号化的内容却用文字描述（大于～、～以上、不足～等）

到了实现阶段，这类描述难免被开发者加入个人的理解，产生歧义，进而导致最后成品与我们想要的东西不一致。

若想改善上述情况，首先要将表达同一个事物的用语统一，坚决不用其他说法来表述该事物。用语的统一能带来认知的统一。需要清单化的东西一定要清单化，明确描述出所有项目。清单最好集中写在一处，其中在需要参考某些资料的部分写明相应资料的位置，以便将来出现变更时能轻松修改。

数值增减的幅度、上下限也要尽量写明。因为它会给是否限制输入内容、项目的显示宽度等设计方面带来影响。

能数值化、符号化的东西尽量用数值、符号来表达，不要用文字描述。因为文字描述更容易产生误解和歧义。

为避免开发者加入个人的理解，写文档时应避免加入多余内容，同时要将必备内容表述得尽量清楚。在发现模棱两可的表述时，如果只是单一的局部问题，我们只需即刻通知其他相关的开发人员，对该部位进行修正即可。但是当发现该问题散布在文档各处时，就要提起注意了。首先查看手头的其他文档，如果这些文档也有同样问题，说明编写文档的前期讨论做得不够充分。此时需要警示项目中的所有人员，因为该问题搞不好会影响整个项目的进程。

【消除模糊不清的描述】

- 表述同一事物时统一用词，使表述有整体感
- “○○等”“○○左右”等描述要列出所有可能性 → 例如 A. ○○、B. □□、C. △△
- 能数值化、符号化的就不用文字描述 → 用“>”“<”“≥”“≤”等符号以及具体数值替换文字描述
- 注明数值增减的幅度和上下限

◉ 明确区分最新版本和非最新版本

存在多个同名（近似名称）文档会带来混乱。相信很多人见过下面这种文件名。

- ○○○○修正版
- ○○○○最终版
- 新○○○○

这些文档的描述往往大同小异，却都各自存在一些需要修正或需要更新的地方。要解决这个现象，最好是对所有同类文件的描述进行一个汇总，做成一个最新的文档，其他的则都视为旧文档全部处理掉。

首先把可以确定的旧版本文档删除。接下来把重复的文档汇总，选出其中更新日期最近、名字最相似的两个进行比对，详细记录二者的区别，在其中一个文件中进行勘误及汇总。汇总完成后找出下一个最相近的文档重复上述工作。所有重复文档的对比、汇总完成之后，将用作比较对象的文档全部归档到一起，按照陈旧文档处理。一定要明确区分最新版本和旧版本的保存位置，防止再发生相同情况。像管理源码一样用版本控制系统来管理文档也是个不错的选择。

从防止同样情况再次发生的角度讲，如果陈旧文档已经明确失去了使用价值，可以考虑直接删除。

【保持文档处于最新状态】

- 删除确实陈旧的文档
- 根据更新日期和标题选择相近的文档逐一汇总
- 做出最新版本的文档之后，其余文档全部视为陈旧文档处理
- 确认没有问题的情况下删除陈旧文档

如果各个文档的审查工作正常进行，则能给后续阶段奠定可靠的基石，保证在遇到问题时有据可依。希望各位在开始编写代码之前先腾出一部分时间处理文档，它能让后续的时间安排更加灵活高效。

13.2.2　测试设计的编写方法（输入与输出）

测试设计的编写完成时间最晚不能晚于功能的实现。

进行测试设计时，需要根据测试内容和目的不同编写不同的输入文档。这是为了对程序进行多角度测试，从而尽量减少程序存在的缺陷。

文档	测试	测试内容 / 目的
功能需求	系统测试（又称综合测试）	最终成品严格按照需求运行
非功能需求	性能测试、压力测试（又称负荷测试）	使用者使用时感觉不到压力
基本设计	集成测试（内部集成 / 外部集成）	各功能之间是否正常协作
详细设计	UT 或单元测试	当前功能是否严格符合要求
（实现）	（试运行）	功能是否如开发者预期正常实现

这个时候，要注意测试设计的编写是否顺利。因为测试的实施项目要以各文档内记录的内容以及正常状态为依据来编写，所以这一过程中能明显看出文档的好坏。测试设计的编写轻松

且顺利，证明文档质量高，反过来如果编写测试设计时频频遇到难题，则说明文档不合格。要是文档中多处出现 13.2.1 节提及的问题，最好重新审查一遍文档。

编写测试设计时常出现的问题有以下几个。

【问题】

- 不知该写什么，写多少
- 测试设计的时间不够用
- 文档记录的内容有问题
- 没有可用作依据的输入文档

如果问题出在文档自身，那么只需要像上面所说的那样，在开始编写测试设计之前重新查看并审查文档即可解决问题。就算没有充足时间，也尽量不要在没更新文档的情况下直接根据最新状态编写测试设计。因为这样一来我们就失去了客观说明测试设计的依据。更新文档可以让所有相关人员及时共享最新信息，因此千万怠慢不得。

接下来我们逐个解决剩下的问题。

◉ 明确测试的目的

如果测试的项目只求详细，那想写多少项目都能写出来（当然也因文档而异）。同样，项目想删怎么都能删掉。所以在设计测试时要注意“必须保证的点（= 测试的目的）”，区分重要的项目和不重要的项目。

测试设计既要完成其在所属流程内应当完成的所有任务，又不能重复记录属于其前后流程的项目。一定要清楚自己编写的测试设计所负责的阶段以及需要完成的任务。

另外，对于用文字难以表述的测试设计，用另附表格的方式往往能轻松地表述清楚。需要记录多种情况时，可以在测试设计文本中写操作流程，然后另附表格来说明各个情况，这样既可以让文档看起来干净利索，又能有效防止遗漏。总而言之，就是不要拘泥于定式，选择最符合该项目测试的形式来编写测试设计。

还有，一个项目内要确认的内容必须限制在一个。比如在“点击按钮后信息写入数据库”中，“生成报告”和“各项目的保存内容正确无误”就要分开来写。这样，在实施时能更轻松地判断是 OK 还是 NG。

【写什么，写多少】

- 注意该测试内必须保证的点（= 测试的目的），区分主干和枝叶
- 不写属于其他测试中要保证的项目（防止重复）
- 难以用文字表述的另附表格
- 一个项目内需确认的内容限制在一个

◉ 从重要的部分开始写

测试设计是基于文档的，那么一旦文档编写的进度落后，势必会压缩测试设计的时间。很多时候，留给测试设计的日程安排都是非常紧的。

这里重点需要注意的地方和上面有些类似，也就是“写什么”“写多少”“能不能不写”的问题。越是时间紧的时候，越需要仔细阅读文档，然后从重要的部分开始写测试设计。如果动手之前不先读一遍，写到一半肯定会遇到瓶颈。因为我们先写了重要的部分，所以最后就算时间不够用了，我们也能保证重要的部分不出问题。如果真的完全没有时间可用，至少也要把需要检查的观点和方针逐条记录下来，这必然会对实施阶段有所帮助。

【时间不够用的时候】

- 仔细阅读文档
- 从重要的部分开始写
- 就算没时间做测试设计，也要将观点和方针逐条记录下来

◉ 利用所有可以利用的资源

在规模较小或人数较少的开发现场，不编写文档、文档写出来不更新导致形同虚设的情况时有发生。

在这种情况下，我们需要一边向相关负责人咨询所需信息一边推进测试设计。如果此时已经有了能够勉强运行的程序，最好一并拿来作参考。咨询来的信息一定要记录下来，哪怕只是草草作个笔记也好，这些可以共享的资料在后续阶段必然会用得到。每次咨询之后都要做个总结，这能避免我们为了同一件事反复去询问负责人。

在没有文档可用的情况下，如果能认真地编写测试设计，测试设计将成为最后进行规格核对时的重要资料。

【没有文档的时候】

- 观察已能运行的程序，掌握大致情况
- 必要信息直接咨询相关负责人
- 把咨询到的信息记录下来，汇总保存
- 要比有文档时更用心地编写测试设计

总而言之，编写测试设计时要注意我们上面提到的所有问题，同时综合考量程序的健壮性、发布前的日程安排等因素，在此基础上选择与测试各阶段方针相符的粒度，根据所选粒度最终完成测试设计的编写。

还有，由于需求变更时必须同时修改测试设计，所以测试设计要尽量选用方便添加、删除操作的格式，免得修改时遇到麻烦。

13.2.3　测试的实施与测试阶段的轮换（做什么，做多少）

本部分所讲的问题通常出现在正式开始测试的时候。实施测试时需要同时准备相关文档以及测试设计。

开始实际测试程序时常出现的问题有下面几个。

【问题】

- 没时间实施所有测试
- Bug 太多，测试难以进行
- 不清楚在发现 Bug 时如何报告
- 没有测试设计

下面我们来对这些问题进行逐条改善。

◉ 实施测试时也要有优先顺序

有些时候，我们不缺文档和测试设计，但交付期前紧迫的时间让我们无法完成数量庞大的全部测试项目。从整个项目的优先顺序来讲，人们往往会先照顾交付期，把执行全部测试项目放到其次。遇到这种情况我们应该怎么办呢?

首先，要确认交付期是否真的无法延后。缩短测试期意味着可以保证的项目相应减少，这很有可能导致发布后出现 Bug 和安全问题，直接影响信誉。

我们需要从整个项目的角度出发，重新认真衡量一下得失，确认交付期是否真的优先于上述几点。

如果考虑到项目整体平衡而需要删除某些测试项目，那么在选择项目前最好先按照以下条件对测试项目进行梳理及分类。

- 已经完成的 / 尚未完成的
- 不完成就无法发布的 / 就算没完成也可以删除而不影响发布的
- 必须详细进行测试的 / 大概确认一下就可以的

以上述分类标准为参考，给各项目设置优先顺序如下。

- 等级 A：已经实现，且不完成就无法发布的功能
- 等级 B：尚未实现，但不完成就无法发布的功能
- 等级 C：大概确认一下就可以的 / 实在不行删掉也可以的功能

A 和 B 是发布所需的最低限度的功能，因此如果它们的测试不够充分，我们连最低限度的功能都无法保证。这两类项目的测试时间一旦无法保证，必须同交付方进行交涉。

如果无论如何都无法延期，则需要将测试能细分的细分，不能细分的调整粒度（= 缩小保证范围，添加限制。此时需要保证整个团队达成共识）。

C 部分就是与时间赛跑，先把这些测试按优先级排成一条线，然后逐个实施。这里要做好心理准备，因为排在后半部分的项目多数时候没时间实施。

确定优先顺序时要避免模糊不清的等级制，一定要严格排序。这是为了保证那些必不可少的测试能够先执行完。这里一旦用了模糊不清的等级制，等到火烧眉毛的时候往往会出现“全都是 A”的情况，或者冒出来“AA 级”“S 级”之类原先根本不存在的等级。这就使当初的分级失去了意义。确定优先顺序是为了给测试项目划分界限（这里界限的标准不是“哪些需要保证”，而是“哪些可以不用保证”），以备万不得已之时可以将没必要的东西砍掉，因此决不能有半点模糊。

制定方针、共享方针之后，剩下的就是跟时间赛跑，在剩余时间内做到最好了。

关于优先顺序再补充一点，性能、压力测试等非功能测试要在环境具备一定条件时尽早实施。因为如果拖到一部分功能完成后再做，出现问题时需要排查的范围将会很大，修正 / 重新测试的周期也会很长。

另外，这些测试要在测试实施的各阶段定期进行。这听起来虽然与前面的言论自相矛盾，但要知道，非功能需求和功能需求一样，有些问题只有在集成之后才会显现出来。

【时间不够用的时候】

- 确认是否真的无法延期
- 确定优先顺序，先测试必不可少的项目
- 优先级较低的项目排成一条直线逐个执行（万不得已时直接放弃）
- 非功能需求的测试要尽早地、持续地实施

◉ 问题过多时先着眼大局

有时我们会遇到这种情况：开始测试之后，程序到处都是毛病根本无法运行，或者按日程表来说理应完成的功能却不能用，结果导致安排好的测试项目无法推进。这种时候就要暂时放下测试，先把已经完成的功能和未完成的功能分个类，做成清单共享给团队。

实现阶段的进度一旦与认知出现偏差，很可能发展成影响整个项目进度的问题，因此需要尽早做到信息共享。如果功能已经实现，但是品质方面存在问题，则需要确定问题的优先顺序并逐个检查。出现问题时眼光不能局限在一个功能内，要尽量站在全局角度。

① 安全方面的问题

② 功能是否正常运行

③ 数据的显示、读写是否正常

④ 设计方面的问题

测试时，先像上面这样大致列出几个重点，把当前功能中优先级高的项目测试完，然后暂时跳过优先级低的项目，直接开始下一个功能的测试。等到所有优先级高的项目测试完后，再回过头来逐个解决优先级低的项目。

【缺陷过多的时候】

- 区分有缺陷的功能和未实现的功能，将未实现的部分做成清单共享出来
- 给要实施的测试项目制定优先顺序
- 目光不局限在一个功能内，测试尽量照顾到整体

◉ 缺陷报告要简洁、详细

如果在测试过程中发现缺陷，应该如何报告呢？如果只说“无法运行”“运行不正常”，负责修改的人也无从改起。所以要尽量加入一些详细信息。

- 实施时间
- 实施环境
- 实施者（负责实施的人，或者当时登录系统的用户等）
- 实施意图
- 出现的问题（与预想结果的偏差）

这些都是在重现问题或者检查问题现状时必不可少的信息。将实施时间加入报告是为了让负责人能通过程序日志进行调查。因为有些问题只有在特定时间才会发生。

报告里还要尽量留下证据。另外，执行测试时的输入值、执行前后数据库的状态最好也保存下来。这些能在发生问题时大幅减轻调查的压力。能否还原发生问题时的情况，直接影响着解决问题的速度。

遇到那些觉得不对劲，但是又无法根据文档判断是否为缺陷的问题时，也要作出报告。借助整个团队的力量解决问题要远比一个人伤脑筋来得快。另外需要注意，出现这种难以判断的情况时，意味着设计阶段很可能潜藏了大问题。

【发现缺陷的时候】

- 尽量详细记录情况并报告
- 尽可能地留下证据
- 无法判断是否为缺陷时，不要急着独自下判断，先报告

◉ 即便没有测试设计，也要保证方针共享

根据项目规模和发布前的日程安排，我们有时候会大胆放弃测试设计，或者由于时间太紧没能完成测试设计。

学习测试设计的时候我们学习过，当剩余时间真的不够用的时候，至少要保证观点或方针明确。在此基础上，再根据实施时的优先顺序，从一旦发生缺陷可能引发严重问题的地方开始测试。如果开始测试后才发现问题比想象中复杂，难以解决，至少要把需确认的项目逐条记录下来。

【没有测试设计的时候】

- 确认、共享观点和方针
- 优先处理最可能引发问题的地方
- 对于太过复杂难以测试的地方，逐条列出需确认的项目

由于实施测试的阶段位于整个项目的后半部分，所以往往需要与时间赛跑。但即便时间不够用，我们也要有计划地推进测试，力求提高效率，用最少的劳力换来最大的效果。

13.3　小结：测试并不可怕

本章我们讲了在项目早期阶段导入测试观点能带来的几点改善。

说得笼统一点，就是确定一个正确的状态，检查当前状态与正确状态之间有多少偏差，然后考虑怎样做能修正这些偏差。至于思路和需确认的事项等，很多都可以直接拿到编程的过程中重复利用。测试能够担保的东西其实并不多，充其量不过是在有限的范围内做个保证，比如“能防止这类错误”“这种情况在预料之中”等，不可能保证完全的正确性。测试能够证明“这个缺陷已被修复”，却不能证明“这里已经没有缺陷了”。

另外，设计者、实现者都是活生生的人，难免犯错和疏忽。所以，无论我们在开发过程中多么谨慎，也不可能将出现问题的可能性降为零。

测试要有计划性。无计划的测试不但浪费时间，获得的成效也低。相反地，有计划的测试能帮助我们提高程序的品质。由于往往到了项目后半期，测试才受到重视，所以人们的注意力常会被实现代码的过程吸引，忘记去关注测试。但是我们希望各位能在项目的初期就去有意识地兼顾测试，放开胆子与测试携手同行，从而提高开发的品质。

第 14 章　轻松使用 Django

基于 Python 开发的 Web 应用框架有很多，例如 Pyramid、Flask、Bottle、Tornado 等。本章我们将目光放在 Django 上，首先对其进行简单学习，然后了解一些在实际开发过程中对 Django 有辅助作用的库。

14.1　Django 简介

Django 是一款基于 Python 开发的全栈式一体化 Web 应用框架。2003 年问世之初，它只是美国一家报社的内部工具，2005 年 7 月使用 BSD 许可证完成了开源。其目的是削减代码量，简单且迅速地搭建以数据库为主体的复杂 Web 站点。它是全栈式框架，因此安装起来很简单，而且使用者众多。这使得 Django 除具有完备的官方文档之外，还有大量的关联文档、丰富的第三方库可供使用。与其他框架相比，Django 用起来要轻松得多。

14.1.1　Django 的安装

Django 的安装与其他 Python 程序包一样，需要通过 pip 进行。

```
$ pip install django
```

本书使用了 Django 的 1.7.1 版。

14.1.2　Django 的架构

Django 继承并简化了 MVC 架构。MVC 中的 Controller 部分基本全由 Django 完成。View 部分则被分割为两部分，即负责 HTML 渲染的模板和负责显示逻辑的视图。所以 Django 又被称为 MTV（Model-Template-View）框架。这个 Django 框架除了 MTV 框架的核心部分，即 O/R 映射工具、URL 分配器（Dispatcher）、视图、模板系统之外，还有管理界面、缓存系统、国际化支持、表单处理等机制和功能。

使用 Django 开发 Web 应用站点时，需准备一个承载着 Django 实例及数据库设置等内容的工程，然后通过在该工程中新建几个应用或者调用外部应用，或者将二者结合起来进行开发。由于每个应用的本质都是 Python 程序包，所以只要按功能（模型、视图、模板等）对这些包进行分离，完全可以拿到其他工程中重复利用。

图 14.1 是 Django 架构处理请求的流程。

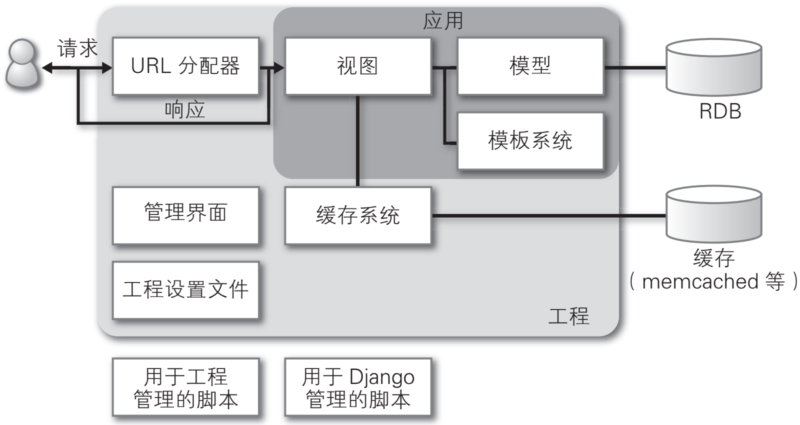


图 14.1　Django 的架构

① 客户端发来的 HTTP 请求被视为 Django 的请求对象

② URL 分配器负责搜索并调用被请求的 URL 所对应的视图

③ 被调用的视图视情况使用模型或模板生成响应对象

④ 响应对象作为 HTTP 响应发回给用户

接下来我们按顺序了解一下 Django 的组成元素。

◉ 工程

工程是承载了 Django 实例的所有设置的 Python 程序包。大部分情况下，一个 Web 站点就是一个工程。工程内可以新建及存放该工程固有的应用，或者保存 Web 站点的设置（数据库设置、Django 的选项设置、各应用的设置等）。

如果完整安装了 Django，可以使用 Django 管理脚本 django-admin 的 startproject 命令创建工程目录 myprj 和初始文件。

```
$ django-admin startproject myprj
```

myprj 的布局如下所示。

```
myprj/
├── manage.py
└── myprj
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

◉ 应用

对于 Django 而言，应用指的是表示单一功能的 Web 应用的 Python 程序包。由于其实质就是 Python 程序包，因此放在 PYTHONPATH 有效的任何位置都没有问题。这里最好尽量减少应用与工程、应用与其他应用之间的依赖关系，做到功能独立，以便在其他工程中重复利用。

比如我们要创建名为 polls 的应用，就可以使用刚刚在工程目录下生成的工程管理脚本 manage.py，执行 `startapp` 命令。

```
$ cd myprj
$ python manage.py startapp polls
```

随后会按照下列布局生成 polls 目录及初始文件。

```
polls/
 ├── __init__.py
 ├── admin.py
 ├── migrations
 │     └── __init__.py
 ├── models.py
 ├── tests.py
 └── views.py
```

◉ 模型

Django 提供了 O/R 映射工具，因此可以用 Python 代码来描述数据库布局。

每个模型都是继承了 django.db.models.Model 类的 Python 的类，分别对应数据库中的一个表格。通过将数据库的字段、关系、行为定义为模型类的属性或方法，我们可以使用丰富且灵活的数据库访问 API。

◉ URL 分配器

URL 分配器机制使得 URL 信息不再受框架及扩展名的制约，从而让 Web 应用的 URL 设计保持简洁。

URL 在 URLconf 模块中进行描述，URLconf 模块中包括使用正则表达式书写的 URL 和 Python 函数的映象。URLconf 能够以应用为单位进行分割，因此提高了应用的可重复利用性。另外，我们可以利用给 URL 设置名称并定义的方式让代码和目标直接通过该名称调用 URL，从而将 URL 设计与代码分离。

◉ 视图

Django 的视图是一类函数，它能够生成指定页面的 HttpResponse 对象或像 Http 404 这样的异常情况，返回 HTTP 请求。典型的视图函数的处理流程通常是先从请求参数中获取数据，读

取模板，然后根据获取到的数据渲染模板。

◉ 模板系统

在 Django 的概念中，模板系统只负责显示，并不是编写逻辑代码的环境。因此 Django 的模板系统将设计与内容、代码分离开来了，是一种功能强、扩展性高、对设计者很友好的模板语言。

模板基于文本而不是 XML，因此它不但能生成 XML 和 HTML，还能生成 E-mail、JavaScript、CSV 等任意文本格式。

另外，如果使用模板继承功能，子模板只需要将父模板中预留的空位填满即可。我们在编写模板时只需要描述各个模板独有的部分，因此可以省去重复冗余的编码过程。

◉ 管理界面

大多 Web 应用在运行过程中，都需要一个专供拥有管理员权限的用户添加、编辑、删除数据的界面，但是实际制作这个界面并不容易。

Django 只需将已完工的模型添加到管理站点，就能根据模型定义动态地生成面，为我们提供一个功能齐全的管理界面（图 14.2、图 14.3）。



图 14.2　登录管理界面后的首页

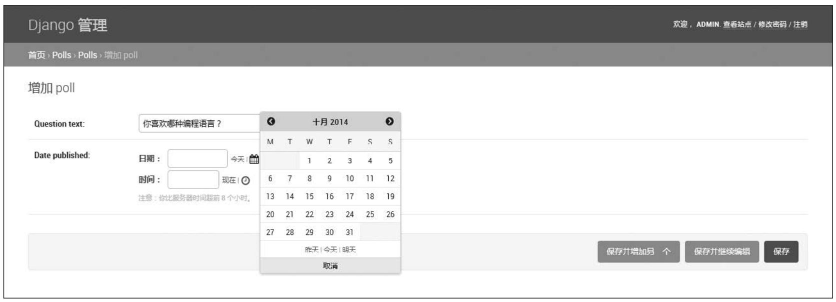


图 14.3　管理界面的模型编辑界面

◉ **缓存系统**

Django 可以使用 memcached 等缓存后端（Cache Backend）轻松地缓存数据。比如可以将动态页面的渲染结果（部分或全部）缓存下来，等到下次需要时直接读取缓存，从而不必每次都对动态页面进行处理。

缓存的后端可以从 memcached、数据库、文件系统、本地内存等位置中进行选择。缓存对象也支持整个网站、特定的整个视图、部分模板、特定数据等。

14.1.3　Django 的文档

在下述网站上可以阅览 Django 1.7 ～ 1.10 以及 dev 版的英文文档。

Django documentation
`https://docs.djangoproject.com/`

从下一节开始，本章的内容均面向有 Django 使用经验的读者，因此，在进入下一节的学习之前，请先阅读文档内的教程部分。

Django 教程
`https://docs.djangoproject.com/en/1.7/intro/tutorial01/`

14.2　数据库的迁移

14.2.1　什么是数据库的迁移

数据库的迁移（Migration）是对数据库中的模式定义以及数据转移进行管理的功能。它的主要作用是版本管理以及版本升级、回滚等，类似于 Ruby 的 Ruby on Rails 框架。

那么，在什么场合下才能体会到它的便捷之处呢？我们经过亲身实践，认为它在以下场合能发挥极大作用。

◉ **多人持续开发时**

多人共同开发的时候，应用程序经常要反映别人的修改，所以模式升级和回滚的机会也相对较多。

◉ 需要在运行过程中修改数据表时

如果需要在运行过程中修改数据表，就必须修改模式以防止现有数据受损。另外，一旦修改后出现问题，还需要立刻进行恢复。这种时候就必须做好版本管理，保证随时可以回滚。

14.2.2 Django 的迁移功能

◉ Django 数据库迁移

Django 的迁移功能支持给模型添加新的字段、向数据库的列添加 null=Ture 等。在版本管理方面，Django 以模型为基准，通过生成并执行迁移文件来完成数据库版本的更新，从而实现版本管理。

迁移文件同时也是从零新建数据库的方法之一。第一个迁移文件的执行结果是从空模式迁移为初始状态的数据表。等到所有迁移文件执行完毕之后，我们就会得到应用了最新版本的数据库模式。另外，如果有上一个版本的数据库，那么只需执行最后生成的迁移文件即可得到最新的数据库模式。

◉ 示例（对新建的应用套用迁移时）

接下来我们以新建应用的情况为例来学习一下迁移功能。首先新建一个应用（LIST 14.1）。

LIST 14.1 新建 polls 应用

```
$ python manage.py startapp polls
```

新建 polls 应用之后，首先在 settings.py 的 INSTALLED_APPS 中添加 polls，然后在 polls/models.py 中创建模型（LIST 14.2）。

LIST 14.2 myprj/settings.py

```
INSTALLED_APPS = (
    ...

    'polls',
)
```

polls 应用需要投票项目（poll）和选项（choice）两个模型，这里我们只创建 poll。poll 中要包含题目（question_test）的信息（LIST 14.3）。

LIST 14.3 polls/models.py

```
# -*- coding: utf-8 -*-
from django.db import models

```

```
class Poll(models.Model):
    question_text = models.CharField(max_length=200)
```

接下来要做的是生成迁移文件。迁移文件通过 `makemigrations` 命令生成（LIST 14.4）。

LIST 14.4　生成第一个迁移文件

```
$ python manage.py makemigrations polls
Migrations for 'polls':
  0001_initial.py:
    - Create model Poll
```

执行上述命令后，会生成 polls/migrations/0001_initial.py 迁移文件。此时数据库中还没有 polls，所以这里我们需要通过 `migrate` 命令执行迁移文件，从而生成 polls（LIST 14.5）。

LIST 14.5　执行 migrate

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, polls, auth, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying polls.0001_initial... OK
  Applying sessions.0001_initial... OK
```

此时会发现 poll 还缺少公布日期（Publication Date），于是我们还需要为它添加公布日期（LIST 14.6）。

LIST 14.6　polls/models.py

```
# -*- coding: utf-8 -*-
from django.db import models

class Poll(models.Model):
    question_text = models.CharField(max_length=200)
    # 添加公布日期
    pub_date = models.DateTimeField('date published')
```

此类细微修改也能通过迁移功能完成。做修改时也要用 `makemigrations` 命令生成迁移文件（LIST 14.7）。

LIST 14.7　生成用于修改的迁移文件

```
$ python manage.py makemigrations polls
```

```
You are trying to add a non-nullable field 'pub_date' to poll without a default;
we can't do that (the database needs something to populate existing rows).
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows)
 2) Quit, and let me add a default in models.py
```

此时，计算机表示“pub_date 中没有设置 default”，询问如何处理已有数据。选择 1 可以向已有数据输入要设置的值，选择 2 则可以中断迁移文件的生成。这里我们选择 1。计算机告诉我们可以使用 datetime module 和 Django 的 timezone mudule，这里我们选择 `timezone.now()`（有时区概念的当前时间。向数据库中添加的值为 UTC）（LIST 14.8）。

LIST 14.8　设置 default 数据

```
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can do e.g.
 timezone.now()
>>> timezone.now()
Migrations for 'polls':
  0002_poll_pub_date.py:
    - Add field pub_date to poll
```

到此，用于修改的迁移文件生成完毕。只生成迁移文件并不能将修改反映到数据库中，因此别忘了执行 `migrate`（LIST 14.9）。

LIST 14.9　执行 migrate

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, polls, auth, sessions
Running migrations:
  Applying polls.0002_poll_pub_date... OK
```

这样一来，修改就反映到数据库中了。

以上就是迁移的基本使用方法。只要记住 `makemigrations` 和 `migrate` 这两个命令，就能够应付绝大部分情况。不过例外总还是有的，为此我们来简单了解一下各个命令。

◉ 命令简介

○ migrate

用于执行迁移文件的命令（LIST 14.10、LIST 14.11）。

LIST 14.10　指定应用执行

```
$ python manage.py migrate application
```

LIST 14.11　指定文件名或编号执行

```
$ python manage.py migrate application file_name_or_number
```

该命令可以指定应用或文件名来执行。如果不指定，则会执行所有应用尚未执行过的迁移文件。

- --fake

  该选项可以将未执行的迁移文件标记为已执行，但不实际执行该文件。比如存在编号很旧的未执行的迁移文件，但当前的数据库状态下执行该文件会出现错误时，就可以使用这个选项。

○ makemigrations

用于生成迁移文件的命令。

- --empty

  用于生成空迁移文件的选项。在我们想手动编写迁移文件的内容，或者需要生成用于数据迁移的文件时，可以使用该选项。关于数据迁移的详细内容，我们将在“数据迁移”部分学习。

○ sqlmigrate

该命令可以查看套用迁移文件时执行的 SQL（LIST 14.12）。

LIST 14.12　查看执行的 SQL

```
$ python manage.py sqlmigrate application file_name_or_number
```

◉ 数据迁移

学习 `makemigrations` 的 --empty 选项时我们了解到，Django 可以借助迁移功能做数据迁移。数据迁移是伴随着值的变化的数据库模式变更，为与一般的模式迁移作区分，才被称为数据迁移。下面我们试着做一个让 poll 的公布日期延后一天的数据迁移（LIST 14.13、LIST 14.14）。

LIST 14.13　生成用于数据迁移的文件

```
$ python manage.py makemigrations --empty polls
Migrations for 'polls':
0003_auto_20141104_0236.py:
```

LIST 14.14　0003_auto_20141104_0236.py

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models, migrations

class Migration(migrations.Migration):

    dependencies = [
        ('polls', '0002_poll_pub_date'),
    ]

    operations = [
    ]
```

生成的迁移文件中只有类的定义和空的方法，并没有描述实际的处理。我们只需添加数据迁移所需的处理，即可用它来做数据迁移（LIST 14.15）。

LIST 14.15　添加让公布日期延后一天的数据迁移

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
import datetime

from django.db import models, migrations

def forward_pub_date_by_one_day(apps, schema_editor):
    Poll = apps.get_model('polls', 'Poll')
    for poll in Poll.objects.all():
        poll.pub_date += datetime.timedelta(days=1)
        poll.save()

def backward_pub_date_by_one_day(apps, schema_editor):
    Poll = apps.get_model('polls', 'Poll')
    for poll in Poll.objects.all():
        poll.pub_date -= datetime.timedelta(days=1)
        poll.save()

class Migration(migrations.Migration):

    dependencies = [
        ('polls', '0002_poll_pub_date'),
    ]
```

```
    operations = [
        migrations.RunPython(forward_pub_date_by_one_day,
                             backward_pub_date_by_one_day)
    ]
```

添加完处理之后，执行 `migrate`，将其反映到数据库中（LIST 14.16）。

LIST 14.16　执行 migrate

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, polls, auth, sessions
Running migrations:
  Applying polls.0003_auto_20141104_0236... OK
```

想了解数据迁移的更多内容可以参考 Django 的迁移文档[①]。

◉ 迁移文件的 squash

随着迁移的频繁使用，迁移文件会越积越多，执行时间自然越来越长。如果想缩短时间，就需要将多个迁移文件合并成一个，这时可以使用 `squashmigrations` 命令（LIST 14.17）。比如在发布 Web 应用时将所有迁移文件合并成一个，这样一来我们在搭建环境时就只需要执行一个迁移。

LIST 14.17　迁移文件的 squash

```
$ python manage.py squashmigrations polls 0003
Will squash the following migrations:
 - 0001_initial
 - 0002_poll_pub_date
 - 0003_auto_20141104_0236
Do you wish to proceed? [yN]
```

计算机询问是否执行，这里键入 y 并执行（LIST 14.18）。

LIST 14.18　迁移文件的 squash

```
Do you wish to proceed? [yN] y
Optimizing...
  Optimized from 3 operations to 2 operations.
Created new squashed migration /path/to/myprj/polls/migrations/0001_squashed_
0003_auto_20141104_0236.py
  You should commit this migration but leave the old ones in place;
  the new migration will be used for new installs. Once you are sure
```

① https://docs.djangoproject.com/en/1.7/topics/migrations/#data-migrations

```
    all instances of the codebase have applied the migrations you squashed,
    you can delete them.
  Manual porting required
    Your migrations contained functions that must be manually copied over,
    as we could not safely copy their implementation.
    See the comment at the top of the squashed migration for details.
```

执行后生成了名为 `0001_squashed_0003_auto_20141104_0236.py` 的新文件。这个迁移文件相当于之前生成的 0001、0002、0003 的总和。往后在新建的数据库中执行 migrate 时会使用这个新的迁移文件，而在已有数据库中则使用原来的迁移文件。

不过，从执行 `squashmigrations` 时输出的信息来看，我们需要手动修改新文件。这是因为之前做数据迁移时，0003_auto_201141104_0236 是手动添加的。新建数据库进行迁移时不需要进行数据迁移，所以我们可以将数据迁移的相关处理从 0001_squashed_0003_auto_20141104_0236 中删除。打开文件，删除 operations 内的 migrations.RunPython(…)。另外，用于添加 pub_date 的迁移文件中设置的当前时间也要一并删除（LIST 14.19）。

LIST 14.19　0001_squashed_0003_auto_20141104_0236.py

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models, migrations
import datetime
from django.utils.timezone import utc

# Functions from the following migrations need manual copying.
# Move them and any dependencies into this file, then update the
# RunPython operations to refer to the local versions:
# polls.migrations.0003_auto_20141104_0236

class Migration(migrations.Migration):

    replaces = [(b'polls', '0001_initial'), (b'polls', '0002_poll_pub_date'),
 (b'polls', '0003_auto_20141104_0236')]

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Poll',
            fields=[
```

```
                ('id', models.AutoField(verbose_name='ID', serialize=False, au
to_created=True, primary_key=True)),
                ('question_text', models.CharField(max_length=200)),
                # 删除下面第一行，添加下面第二行
                # ('pub_date', models.DateTimeField(default=datetime.datetime
(2014, 11, 4, 1, 22, 22, 729029, tzinfo=utc), verbose_name=b'date published')),
                ('pub_date', models.DateTimeField(verbose_name=b'date published')),
            ],
            options={
            },
            bases=(models.Model,),
        ),
        # 删除以下被注释的部分
        # migrations.RunPython(
        #     code=polls.migrations.0003_auto_20141104_0236.forward_pub_date
_by_one_day,
        #     reverse_code=polls.migrations.0003_auto_20141104_0236.backward
_pub_date_by_one_day,
        #     atomic=True,
        # ),
    ]
```

新建数据库进行迁移时，只需要执行 polls 的一个迁移文件。

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, polls, auth, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying polls.0001_squashed_0003_auto_20141104_0236... OK
  Applying sessions.0001_initial... OK
```

无论是多人持续开发时还是需要在运行过程中修改数据表时，迁移都能大幅减少劳动力的付出以及出现操作失误的风险。各位请务必试一试，亲自体验其效果。

14.3　fixture replacement

14.3.1　什么是测试配置器

在第 8 章中，我们学习了测试的相关知识，但在实际编写测试代码的过程中，有时必须依

赖于模块和数据，这种时候就要事先准备测试数据。

Django 默认能够将测试数据事先读入数据库。Django 的测试配置器是一种专门用来让 Django 读入数据库的特定格式数据文件。除测试之外，测试配置器还可以用来导入开发初期阶段需要用到的数据。

◉ 用 loaddata 命令读取配置器

下面我们手动创建一个用于配置器的数据文件，然后读取它。这里需要利用前面生成的 Poll 模型。我们首先在工程目录下创建名为 polls.json 的文件，文件内容如 LIST 14.20 所示。

LIST 14.20　polls.json

```
[
  {
    "model": "polls.poll",
    "pk": 1,
    "fields": {
      "question_text": "Why not use the fixture",
      "pub_date": "2011-11-01 00:00:00Z"
    }
  },
  {
    "model": "polls.poll",
    "pk": 2,
    "fields": {
      "question_text": "Why not use the fixture2",
      "pub_date": "2011-11-02 00:00:00Z"
    }
  }
]
```

pk 指模型的 primary key 的 ID。接下来将这个数据文件加载到数据库中（LIST 14.21）。

LIST 14.21　执行 loaddata

```
$ python manage.py loaddata polls.json
```

执行完毕后，可以在数据库的 polls_poll 表中看到上述配置器提供的数据。各位可以使用 SQL 等进行查看。

◉ 配置器文件的存放位置

在工程开始阶段，将配置器文件放在工程根目录下并无大碍，但是随着应用规模越来越大，文件自然也越来越多，放在工程根目录下将很难维护。这种时候就需要修改配置器文件的存放位置并进行整理。整理有以下 2 个方法。

- 存放在对应应用的 fixtures 目录下
- 存放在 settings.py 的 FIXTURE_DIRS 指定的目录下

比如我们在应用目录下创建了 fixtures 目录（例如 polls/fixtures/polls.json），我们就可以将该应用需要读取的配置器文件放在这里。这样一来，只需要执行 python manage.py loaddata polls.json 命令即可读取配置器文件。另一种就是设置 settings.py 中的 FIXTURE_DIRS，该变量所指目录下存放的配置器文件将被视为读取对象。如果想把任意目录作为读取对象，建议使用这种方法。LIST 14.22 是添加与 manage.py 同级的 fixtures 目录的方法。

LIST 14.22　FIXTURE_DIRS 的设置

```
FIXTURE_DIRS = [
    os.path.join(BASE_DIR, 'fixtures'),
]
```

◉ 在 TestCase 中使用配置器

Django 的 TestCase 能够自动读取配置器的信息。只要在 TestCase 的属性里设置了 fixtures，所指定的配置器文件就会被自动读取。在 LIST 14.23 中，我们读取了名为 polls.json 的配置器，使得测试用例可以使用配置器内记录的数据。

LIST 14.23　使用配置器的 TestCase

```
from django.test import TestCase
from polls.models import Poll

class PollsTestCase(TestCase):
    fixtures = ['polls.json']

    def setUp(self):
        # 一般的测试定义

    def testPoll(self):
        # 使用配置器的测试
```

◉ 用 dumpdata 生成配置器文件

很多时候，纯手动编写配置器文件并不现实（比如需要大量档案或者多种不同情况的数据时）。Django 有生成 fixutre 文件的辅助功能，那就是 `dumpdata` 命令。`dumpdata` 命令可以用数据库里的数据生成配置器文件（LIST 14.24）。

LIST 14.24　执行 dumpdata

```
$ python manage.py dumpdata polls > polls.json
```

执行完毕后生成了名为 polls.json 的配置器文件。内容与之前用 loaddata 加载的内容相同。

14.3.2　几种不便使用默认配置器的情况

配置器用起来很方便，但编写测试代码时我们也会遇到配置器碍事的情况，下面就举几个例子来说明一下。

◉ 日期字段会变成特定日期

我们再来看看上面提到的 polls.json，会发现 pub_data 列的日期是固定的（LIST 14.25）。

LIST 14.25　polls.json

```
[
  {
    "model": "polls.poll",
    "pk": 1,
    "fields": {
      "question_text": "Why not use the fixture",
      "pub_date": "2011-11-01 00:00:00Z"
    }
  },
  {
    "model": "polls.poll",
    "pk": 2,
    "fields": {
      "question_text": "Why not use the fixture2",
      "pub_date": "2011-11-02 00:00:00Z"
    }
  }
]
```

比如我们的测试代码希望用到包含“今天的日期”的 Poll 模型时，就需要将配置器文件中的 pub_data 修改成“今天的日期”才行。在这种需要用到“今天的日期”或类似数据的情况下，就不适合用配置器来准备测试数据。

◉ 配置器不容易维护

举个例子，我们给 Poll 模型添加新的属性（列）时，需要对 polls.json 的所有数据添加属性。这将是一个非常费劲的工作。特别是在开发新的应用时，模型的修改往往很频繁，要想维护配置器文件使之能跟上模型的变化，成本通常不低。

可见，配置器在事先准备数据方面显得十分便捷，但它准备的数据缺乏灵活性和可维护性。

接下来我们将学习一个能解决上述这些问题的工具——factory_boy。

14.3.3　如何使用 factory_boy

factory_boy[①] 可以帮助我们生成更加灵活、更易维护的配置器。从名字可以看出来，它是 Ruby 经典工具 factory_girl[②] 的 Python 版。

Django 的配置器是基于文件来准备数据，而 factory_boy 是生成对应于数据模型的 Factory 类，因此它能够动态且灵活地为我们准备测试数据。这种不以文件形式提供配置器，而是提供能带来同样效果的数据模型的工具称为 fixture replacement。除支持 Django 外，factory_boy 还支持其他 O/R 映射工具。

这里我们用 pip 安装 factory_boy（LIST 14.26）。

LIST 14.26　安装 factory_boy

```
$ pip install factory-boy
```

本书使用了 factory_boy 的 2.4.1 版本。

◉ Factory

Factory 对于将目标对象实例化时所需的一系列属性进行了定义。在命名 Factory 类时，需要让使用者能够通过我们赋予的名称推测出其目标对象的类（LIST 14.27）。

LIST 14.27　PollFactory 类

```
import factory

from django.utils import timezone

from polls.models import Poll

class PollFactory(factory.django.DjangoModelFactory):
    question_text = 'factory question'
    pub_date = timezone.now()

    class Meta:
        model = Poll
```

◉ 构建策略

factory_boy 支持多种不同的构建策略。构建策略是确定数据模型生成状态的方法。在测试用例中，如果需要使用不同状态的目标对象，就会用到它们（LIST 14.28）。

---

① https://github.com/rbarrois/factory_boy
② https://github.com/thoughtbot/factory_girl

LIST 14.28　支持的构建策略

```
# 返回没有 save 的 Poll 实例
poll = PollFactory.build()

# 返回已 save 的 Poll 实例
poll = PollFactory.create()

# 以字典形式返回生成 Poll 实例时所用的属性
attributes = PollFactory.attributes()

# 返回所有属性桩代码化后的对象
stub = PollFactory.stub()
```

直接实例化 Factory 类时，其运行效果与 `create()` 相同（LIST 14.29）。

LIST 14.29　Factory 类的实例化

```
poll = PollFactory() #=> 与 PollFactory.create() 效果相同
```

生成实例时可以通过传递关键字传值参数的方式覆盖原有属性值，而无需关心使用的构建策略（LIST 14.30）。

LIST 14.30　覆盖属性

```
poll = PollFactory.create(question_text='changed question')
```

◉ 延迟计算属性

虽然我们在定义 Factory 的时候已经添加了静态的属性，但有些属性（关联的数据模型、需要动态生成的属性等）需要在每次生成实例时进行设置。准备这类属性就需要用到 LazyAttribute 了（LIST 14.31）。

LIST 14.31　LazyAttribute 的示例

```
class UserFactory(factory.django.DjangoModelFactory):
    first_name = 'Beproud'
    last_name = 'Taro'
    email = factory.LazyAttribute(lambda a:
                                  '{0}.{1}@example.com'.format(a.first_name,
                                                               a.last_name).lower())
    class Meta:
        model = User

UserFactory().email #=> beproud.taro@example.com
```

◉ 序列

根据特定格式需要用到连续的属性值时，可以使用序列（Sequence）（LIST 14.32）。

LIST 14.32　Sequence

```
class UserFactory(factory.django.DjangoModelFactory):
    email = factory.Sequence(lambda n: 'person{0}@example.com'.format(n))

    class Meta:
        model = User

UserFactory().email  # => 'person0@example.com'
UserFactory().email  # => 'person1@example.com'
```

基本的内容到这里就了解清楚了。接下来我们看看如何用 factory_boy 解决 Django 中的配置器不好用的问题。

14.3.4　消除“不便使用默认配置器的情况”

◉ 解决“日期字段会变成特定日期”的问题

按照 LIST 14.33 所示定义 PollFactory 类，然后为测试用例的属性配上合适的日期，这样我们就能得到想要的数据模型了。

LIST 14.33　设置灵活的日期字段

```
class PollFactory(factory.django.DjangoModelFactory):
    question_text = 'factory question'
    # 配上与测试匹配的日期
    pub_date = timezone.now()

    class Meta:
        madel = Poll

# 还可以在生成实例时设置
poll = PollFactory.create(pub_date=timezone.now())
```

◉ 解决“配置器不容易维护”的问题

配置器不易维护的问题怎么解决呢？ Django 的配置器在给模型添加列时，需要对配置器的所有数据进行修改，但如果使用 factory_boy，则只需要给 Factory 类添加列即可。

只需进行下述修改，所有通过 PollFactory 生成的 Poll 模型中就都添加了 user_name 列（LIST 14.34）。

LIST 14.34　添加列

```
class PollFactory(factory.django.DjangoModelFactory):
    question_text = 'factory question'
    # 添加的列
    user_name = 'factory san'
    pub_date = timezone.now()

    class Meta:
        model = Poll
```

通过上面的内容可以看出，Django 的配置器对于简单的测试、事先准备数据等用途而言十分便捷，但要想在实际开发中持续地实施测试，还是建议使用更加灵活且易于维护的 factory_boy。

14.4　Django Debug Toolbar

接下来了解一下在用 Django 开发应用的过程中辅助调试的 Django Debug Toolbar。

Django Debug Toolbar 的简介

用 Django 进行开发时，各位是否想过“显示某个页面的过程中总共发送了哪些 SQL”呢？其实只要使用 Django Debug Toolbar[①]，我们就可以在开发 Web 页面的同时查看“发送了哪些 SQL”“处理花费了多少时间”等信息了。

除了 SQL 之外，使用 Django Debug Toolbar 还可以查看许多在开发过程中想要查看的信息。下面是除 SQL 之外可以查看的项目。

Versions	Django 的版本
Time	显示视图所用的时间
Settings	settings.py 中描述的设置值
Headers	HTTP 请求 / 响应的头信息
Request	GET/POST/Cookie/Session 的变量信息
StaticFiles	静态文件的相关信息
Templates	模板的相关信息
Cache	缓存框架的访问信息
Signals	Django 的内置 signal 信息
Logging	被记录的日志信息

① https://github.com/django-debug-toolbar/django-debug-toolbar

这里我们来了解一下几个可查看的信息。另外，本书使用了 Django Debug Toolbar 的 1.2.2 版本。现在来看看它的具体使用方法，先从安装开始。

◉ 安装

```
$ pip install django-debug-toolbar
```

安装完成之后在 settings.py 中添加如 LIST 14.35 所示的内容。

LIST 14.35　INSTALLED_APPS 内添加的内容

```
INSTALLED_APPS = (
    .
    .
    'django.contrib.staticfiles'  # 在 'debug_toolbar' 之前设置
    .
    .
    'debug_toolbar' # <- 添加
)
```

这里的 `'django.contrib.staticfiles'` 要设置在 `'debug_toolbar'` 之前。另外，Django 1.7 默认设置了 `'django.contrib.staticfiles'`。

◉ 查看 Debug Toolbar

设置完 settings.py 之后重启 Django，随后浏览器上将显示图 14.4 所示的调试用工具栏。

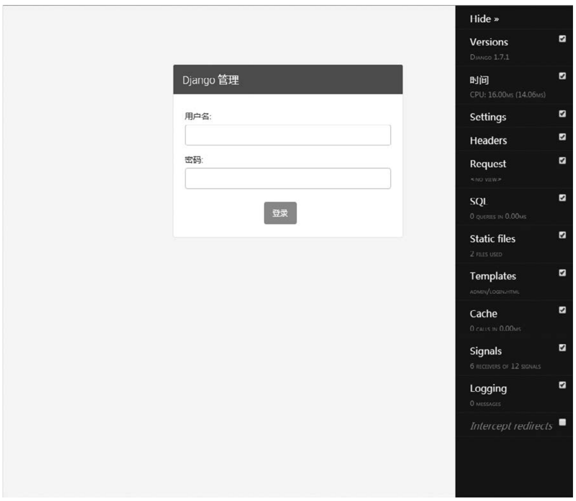


图 14.4　Debug Toolbar

不需要显示 Debug Toolbar 时，点击右上角的 Hide 即可隐藏工具栏。

◉ SQL

如图 14.5 所示，我们能够查看当前视图内发送的 SQL 语句。除此之外还有 SQL 的 Explain 的执行结果以及发送 SQL 前的 Stacktrace 等信息。



图 14.5 SQL

◉ Versions

Versions 会显示当前视图使用的模块的版本信息（图 14.6）。

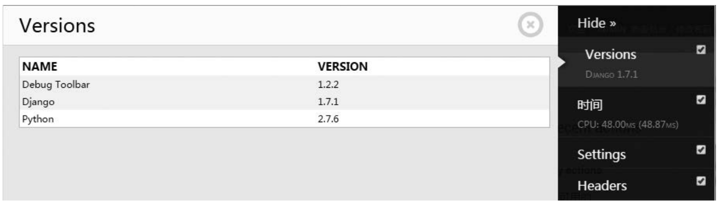


图 14.6 Versions

◉ 时间

如图 14.7 所示，“时间”能查看显示当前视图所消耗的时间。我们来简单看一下其中的主要项目。

- User CPU time：从接到请求到渲染完页面的时间

- System CPU time：从渲染完毕到服务器生成 HTTP 响应并返回给客户端所用的时间
- Total CPU time：User CPU time + System CPU time 的总和，即从接到请求到返回响应所用的时间

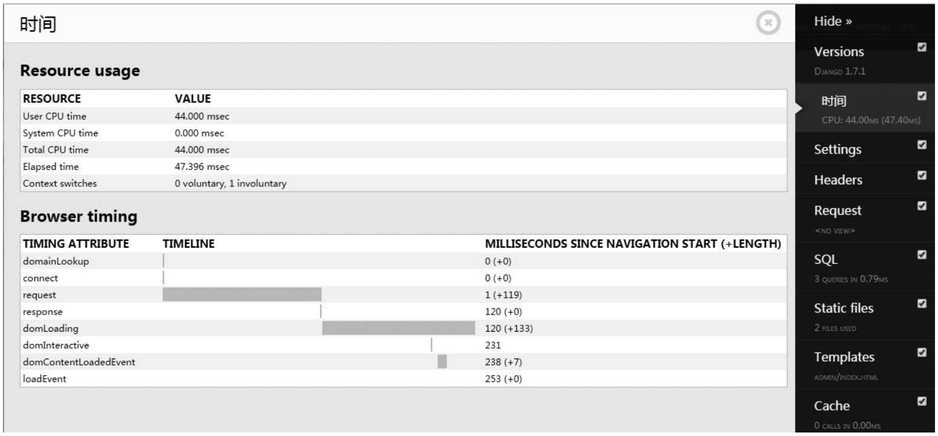


图 14.7　时间

◉ Settings

Settings 可以列表查看 Django 的 settings.py 内设置的值（常量）。这里能查看的只有常量，因此不会显示 settings.py 中定义的函数等内容（图 14.8）。



图 14.8　Settings

◉ Headers

Headers 里可以查看 HTTP 请求 / 响应的头信息（图 14.9）。



图 14.9　Headers

◉ Request

Request 可以查看 HTTP 请求发送的 GET/POST/Cookie 的内容、Session 的内容，以及被调用的视图的信息（图 14.10）。

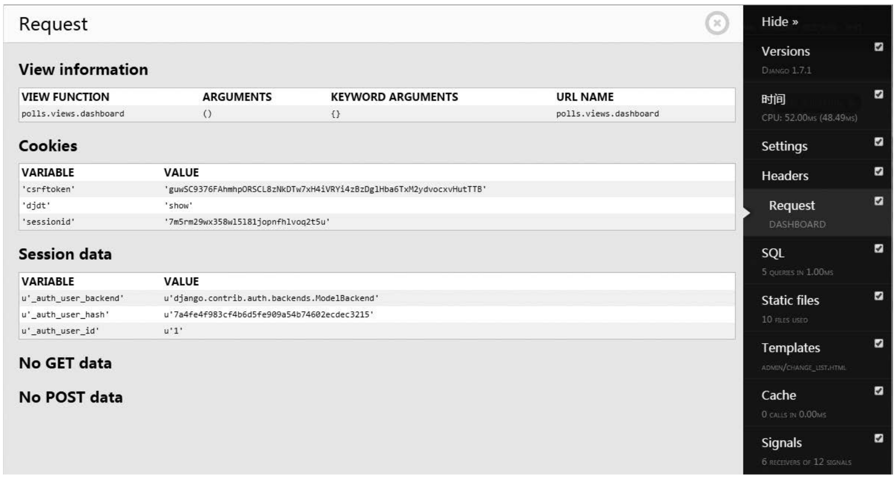


图 14.10　Request

◉ Static files

Static files 可以查看通过 django.contrib.staticfiles 获取的静态文件列表，以及 staticfiles 获取文件时的对象目录列表（图 14.11）。



图 14.11　StaticFiles

◉ Templates

Templates 里可以查看当前视图使用的模板（包含继承关系）以及传给模板的 Context、Context processor 的列表（图 14.12）。

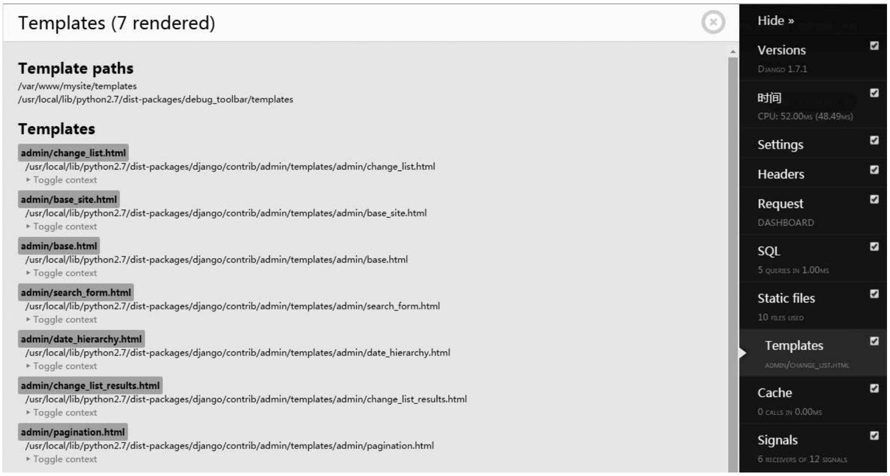


图 14.12　Templates

◉ Cache

Cache 可以查看当前视图使用的缓存后端的列表（图 14.13）。

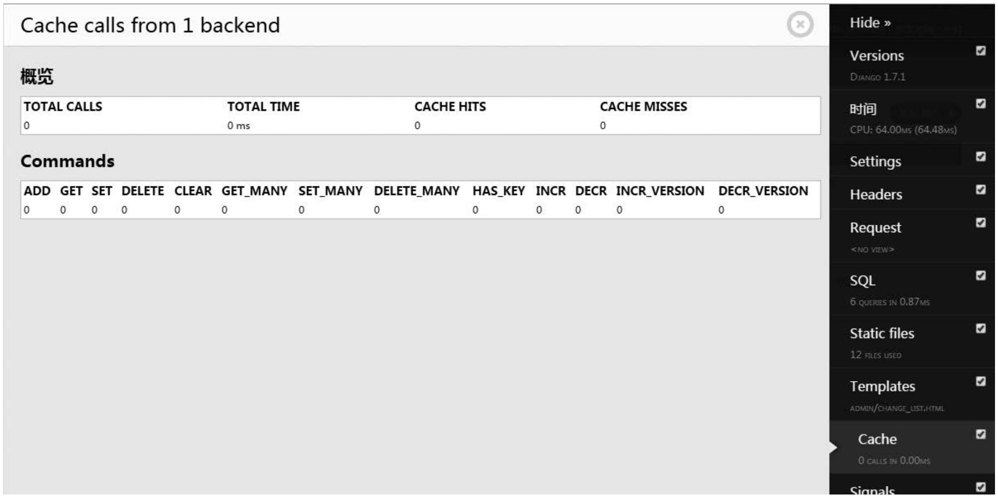


图 14.13　Cache

◉ Logging

假设 polls 应用的视图中输出了如 LIST 14.36 所示的调试日志，那么浏览器上就会显示如图 14.14 所示的日志。

LIST 14.36　Logging 的设置示例

```
import logging
logger = logging.getLogger(__name__)  # __name__ 处代入模块路径 'polls.views'

def detail(request, poll_id):
    poll = get_object_or_404(Poll, id=poll_id)
    logger.info('Poll: %s', poll.id)
    return TemplateResponse(request, 'polls/detail.html',
                            {'poll': poll})
```

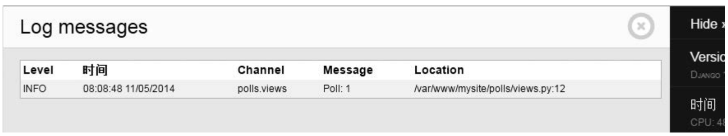


图 14.14　Logging

◉ Intercept redirects

勾选工具栏下方的“Intercept redirects”选项，网页的重定向将被拦截。这项功能可以帮助我们确认重定向的时间点、查看返回重定向响应的视图内都执行了哪些处理。

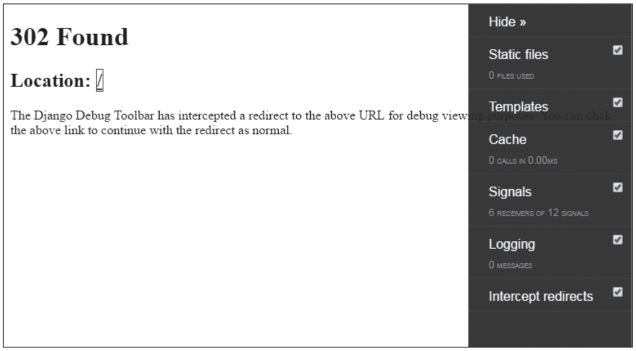


图 14.15　Intercept redirects

◉ 自定义显示

工具栏的显示可以根据开发应用的情况与阶段进行自定义。自定义的方法很简单，只要在 settings.py 中添加 DEBUG_TOOLBAR_PANELS 项，按照自己的需要对其中内容进行重排或改为注释，即可修改工具栏的显示项目（LIST 14.37）。

LIST 14.37　DEBUG_TOOLBAR_PANELS 的设置

```
DEBUG_TOOLBAR_PANELS = [
    'debug_toolbar.panels.versions.VersionsPanel',
    'debug_toolbar.panels.timer.TimerPanel',
    'debug_toolbar.panels.settings.SettingsPanel',
    'debug_toolbar.panels.headers.HeadersPanel',
    'debug_toolbar.panels.request.RequestPanel',
    'debug_toolbar.panels.sql.SQLPanel',
    'debug_toolbar.panels.staticfiles.StaticFilesPanel',
    'debug_toolbar.panels.templates.TemplatesPanel',
    'debug_toolbar.panels.cache.CachePanel',
    'debug_toolbar.panels.signals.SignalsPanel',
    'debug_toolbar.panels.logging.LoggingPanel',
    'debug_toolbar.panels.redirects.RedirectsPanel',
]
```

14.5　小结

本章我们聚焦基于 Python 开发的 Web 应用框架 Django，对其架构进行了学习，并了解了一些有助于开发的库。

如果要开发的 Web 应用很简单，那么默认设置的 Django 也就够我们用了。不过，一旦我

们希望让开发或运营的 Web 应用更加健壮和安全（业务方面尤为多见），“这里做的还不够”“那里再改善一点就好了”之类的需求也就多了起来。正如本章所讲的那样，很多时候，可重复利用的 Django 应用和 Python 库可以帮助我们解决或者一定程度上改善这些问题。

当然，实际开发中方便好用的库远不止本章介绍的这些。下面我们再简单了解几个。

- django-extensions[①]
  管理命令、数据库字段等的结合体，能便捷地为Django框架做功能扩展。
- sorl-thumbnail[②]
  Django应用，通过专用模板标签和管理命令简化缩略图的生成与显示。
- bpmappers[③]
  简化模块映射的Python库。第15章将对其作详细介绍。

① https://github.com/django-extensions/django-extensions
② https://github.com/mariocesar/sorl-thumbnail
③ https://bitbucket.org/tokibito/python-bpmappers

第 15 章　方便好用的 Python 模块

各位在开发应用的过程中有没有想过“这个处理是不是有人实现过”“这个东西该怎么实现”之类的问题呢？这种时候最好先去 Google 上看看是否有现成的模块可用。如果只需要 Python 模块，可以到 PyPI 上去找。发现有能拿来就用的模块绝对是一大好事。确认一下许可证，只要没问题就放心去用吧。

如果能顺利缩短开发时间自然是好事一桩，但现实中往往不那么顺利，比如找不到好用的模块，或是找到模块却不知道怎么用，查询用法又花掉许多时间等。最理想的情况是事先知道这些好用模块的适用场合及使用方法，以便有需要时直接拿来用。因此我们要从平时就注意积累，多通过书籍、网络博客、RSS 订阅等途径收集相关信息。

本章的目的就是充当各位的信息源，为各位介绍一些方便好用的 Python 模块。

15.1　轻松计算日期

日期是大部分系统都要用到的，但是它的计算比较复杂，因此很容易出现 Bug。开发高品质软件时要尽量避免复杂的操作。使用 dateutil 模块可以让我们用简单的描述来完成复杂的日期计算。

dateutil

`http://labix.org/python-dateutil`

15.1.1　日期计算的复杂性

首先我们了解一下日期计算中容易出 Bug 的地方。

◉“1 个月后”是哪一天

遇到“1 月 1 日的 1 个月后是哪一天”的问题时，大部分人都会回答“2 月 1 日”。那么，换成“1 月 31 日的 1 个月后是哪一天”，答案又是怎样呢？恐怕我们会得到“2 月 28 日或 29 日”“3 月 3 日”“2 月 31 日”等多种回答。在开发系统的过程中，如果用了“1 个月后”这种模糊不清的表述，开发者之间很可能产生认识上的分歧，最终开发结果就会出现 Bug。在这个例子中，“30 天之后”“下个月的最后一天”等表述都比“1 个月后”清楚得多，它们能准确指定一个日期，极大地避免歧义。

◉ 求下个月的最后一天

首先我们用 Python 标准模块 datetime 的 timedelta 来编写一个计算“下个月最后一天”的程序（LIST 15.1）。

LIST 15.1　datetime_dateutil_1.py

```
# coding: utf-8
from datetime import datetime, timedelta

def main():
    # 下个月最后一天 = 下下个月的前一天
    now_time = datetime.now()
    # 获取下下个月第一天的日期对象
    first_day_of_after_two_month = datetime(now_time.year,
                                            now_time.month + 2,
                                            1)
    # 获取下个月最后一天的日期对象
    last_day_of_next_month = \
        first_day_of_after_two_month - timedelta(days=1)
    # 输出结果
    print last_day_of_next_month.date()

if __name__ == '__main__':
    main()
```

执行上述代码会得到如 LIST 15.2 所示的结果。

LIST 15.2　执行结果

```
$ python datetime_dateutil_1.py
2012-02-29
```

这里我们通过“下下个月第一天的前一天”算出了“下个月的最后一天”。虽然这乍看上去没有什么问题，但实际上，在求“下下个月的第一天”时，我们忘记考虑年份的更迭了。now_time.month 可以取 1 ~ 12 的值，而这个值加上 2 有可能达到 13、14。因此，这个程序在遇到 11 月和 12 月时会发生例外。我们想要的运行情况是 now_time.month 为 11 和 12 时增算一年。问题修正后的代码如 LIST 15.3 所示。

LIST 15.3　datetime_dateutil_2.py

```
# coding: utf-8
from datetime import datetime, timedelta

def main():
```

```
    # 下个月最后一天 = 下下个月的前一天
    now_time = datetime.now()
    # 获取下下个月第一天的日期对象（考虑跨年问题）
    if now_time.month in [11, 12]:
        first_day_of_after_two_month = datetime(now_time.year + 1,
                                                now_time.month + 2 - 12,
                                                1)
    else:
        first_day_of_after_two_month = datetime(now_time.year,
                                                now_time.month + 2,
                                                1)
    # 输出下个月最后一天
    last_day_of_next_month = \
        first_day_of_after_two_month - timedelta(days=1)
    # 输出结果
      print last_day_of_next_month.date()

if __name__ == '__main__':
    main()
```

可见，日期计算常常会遇到复杂的边界问题，很容易出现 Bug。下面我们用 dateutil 模块来简化这段代码。

15.1.2　导入 dateutil

用 pip 命令安装 dateutil 模块，代码如 LIST 15.4 所示。2014 年 12 月初的 dateutil 最新版本为 2.3。

LIST 15.4　用 pip 命令安装 dateutil 模块

```
$ pip install python-dateutil==2.3
```

使用 dateutil 模块的 relativedelta 函数时，我们可以使用如 LIST 15.5 所示的代码获取前面提到的“下个月最后一天”。

LIST 15.5　datetime_dateutil_3.py

```
# coding: utf-8
from dateutil.relativedelta import relativedelta
from datetime import datetime, timedelta

def main():
    # 下个月最后一天 = 下下个月的前一天
    now_time = datetime.now()
```

```
    # 输出下个月最后一天
    last_day_of_next_month = \
        now_time + relativedelta(months=2, day=1, days=-1)
    # 输出结果
      print last_day_of_next_month.date()

if __name__ == '__main__':
    main()
```

relativedelta 与 timedelta 一样，可以对 datetime 对象进行加减运算。关键字传值参数方面有 day、second、hour 等单数型和 days、seconds、hours 等复数型可供选择。单数型的传值参数为指定数值，复数型的传值参数为指定增减幅度。

◉ rrule

接下来要介绍的是 rrule，它可以获取符合指定规则的日期对象。比如 LIST 15.6，这段代码的作用是获取 2012 年 1 月 1 日到 2012 年 2 月 1 日的周一和周三的日期对象并输出。

LIST 15.6　dateutil_rrule.py

```
# coding: utf-8
from datetime import datetime
from dateutil.rrule import rrule, DAILY, MO, WE

def main():
    # 生成 rrule 对象
    rrule_obj = rrule(DAILY,  # 每天
                      byweekday=(MO, WE),  # 周一、周三
                      dtstart=datetime(2012, 1, 1),  # 2012 年 1 月 1 日起
                      until=datetime(2012, 2, 1))  # 2012 年 2 月 1 日止
    # 逐个取出符合条件的日期对象并显示在屏幕上
    for dt in rrule_obj:
        print dt

if __name__ == '__main__':
    main()
```

rrule 函数的第一个传值参数为获取间隔，可以指定 YEARLY（每年）、MONTHLY（每月）、WEEKLY（每周）、DAILY（每天）、HOURLY（每小时）、MINUTELY（每分钟）、SECONDLY（每秒）。其他条件通过选项指定。以 LIST 15.6 中的代码为例，byweekday 指定了星期几，dtstart 指定了开始日期，until 指定了终止日期。

实际执行结果如 LIST 15.7 所示。

LIST 15.7　执行结果

```
$ python dateutil_rrule.py
2012-01-02 00:00:00
2012-01-04 00:00:00
2012-01-09 00:00:00
2012-01-11 00:00:00
2012-01-16 00:00:00
2012-01-18 00:00:00
2012-01-23 00:00:00
2012-01-25 00:00:00
2012-01-30 00:00:00
2012-02-01 00:00:00
```

15.2　简化模型的映射

近年来，Web 系统为保证服务器与客户端、服务器与服务器之间的协作，越来越多地开始提供 JSON、XML 等格式的 API。在这类 API 的内部处理中，O/R 映射工具生成的对象要序列化成 JSON 或 XML 格式。

开发 API 时，API 提供的 JSON 数据的结构必须与 O/R 映射工具生成的模型对象的结构一致，否则就会出现问题。这种问题称为阻抗失配（Impedance Missmatch）。这种时候，如果模型层级结构比较复杂，那么模型的重复利用、代码的可读性、维护成本等方面都会遇到困难。

这里我们学习一个能有效解决阻抗失配的模块——bpmappers。

> bpmappers
> 　http://bpmappers.readthedocs.io/en/latest/（日文）
> 　https://pypi.python.org/pypi/bpmappers

15.2.1　模型映射的必要性

在实际开发系统的过程中，API 规定的键名与值的对应关系很少能与数据模型的结构一致。接下来，我们以使用 JSON 格式返回响应的 API 为例进行学习。现在假设系统中使用了如 LIST 15.8 所示的 User 类的数据模型。

LIST 15.8　User 类

```
class User(object):
    def __init__(self, id, password, nickname, age):
```

```
        self.id = id  # 用户 ID
        self.password = password  # 密码
        self.nickname = nickname  # 昵称
        self.age = age  # 年龄
```

这个数据模型拥有“用户 ID”“密码”“昵称”“年龄”这 4 个值。而在我们生成的 API 中，只将“用户 ID”和“昵称”两个值包含到响应之中。该 API 通过如下 JSON 格式的响应公开了 User 类的数据。

```
{"user_id": "用户 ID", "user_nickname": "昵称"}
```

接下来写一个函数，使用该函数可以获取一个 User 类的对象，并将其转换为 JSON 格式（LIST 15.9）。

LIST 15.9　将 User 类对象转换为 JSON 格式的函数

```
import json

def convert_user_to_json(user):
    """获取一个 User 对象并返回 JSON
    """
    # 生成用于转换格式的字典对象
    user_dict = {
        'user_id': user.id,  # 使用名为 user_id 的键
        'user_nickname': user.nickname,  # 使用名为 user_nickname 的键
    }
    return json.dumps(user_dict)  # 转换为 JSON
```

这个函数通过 user_dict 变量生成字典对象，它实质上是给模型类的值与字典对象做了映射。像上面这样，我们用 API 提供数据模型的值时，必须给键和值做好映射。

数据模型与 API 响应数据的结构一致时，可以通过给数据模型添加元信息的方式简化映射的描述。使用 O/R 映射工具的数据模型大多含有元信息，因此映射更加简单一些。但正如例子所示，我们很少能遇到数据结构一致的模型，所以描述映射操作是必不可少的一步。

15.2.2　映射规则的结构化与重复利用

在需要返回多种响应的 API 时，意义相同部分的映射代码要保持一致，以便重复利用。

LIST 15.10 是一个返回简单的用户数据以及留言数据（包含用户和文本的数据）的 API。为便于理解，这里不采用 Web API 的形式，而是直接在控制台调用并显示结果。另外，本例中没有使用数据库。

LIST 15.10　mapping_model.py

```
# coding: utf-8
import json

class User(object):
    def __init__(self, id, password, nickname, age):
        self.id = id  # 用户 ID
        self.password = password  # 密码
        self.nickname = nickname  # 昵称
        self.age = age  # 年龄

class Comment(object):
    def __init__(self, id, user, text):
        self.id = id  # 留言 ID
        self.user = user  # 用户 ID
        self.text = text  # 留言内容

def get_user(user_id):
    """ 返回用户对象的函数
    """
    # 实际开发时应该访问数据库
    user = User(id=user_id,
                password='hoge',
                nickname='tokibito',
                age=26)
    return user

def get_comment(comment_id):
    """ 返回留言对象的函数
    """
    # 实际开发时应该访问数据库
    comment = Comment(id=comment_id,
                      user=get_user('bp12345'),
                      text=u'Hello, world!')
    return comment

def mapping_user(user):
    """User 模型与 API 的映射
    """
    return {'user_id': user.id, 'user_nickname': user.nickname}

def mapping_user_2(user):
    """User 模型与 API 的映射 2
```

```
    """
    return {'user_id': user.id,
            'user_nickname': user.nickname,
            'user_age': user.age}

def mapping_comment(comment):
    """Comment 模型与 API 的映射
    """
    return {'user': mapping_user(comment.user), 'text': comment.text}

def api_user_json(user_id):
    """ 以 JSON 格式返回用户数据的 API
    """
    user = get_user(user_id)  # 获取 User 对象
    user_dict = mapping_user(user)  # 映射到字典
    return json.dumps(user_dict, indent=2)  # 以 JSON 格式返回

def api_user_detail_json(user_id):
    """ 以 JSON 格式返回用户详细数据的 API
    """
    user = get_user(user_id)  # 获取 User 对象
    user_dict = mapping_user_2(user)  # 映射到字典
    return json.dumps(user_dict, indent=2)  # 以 JSON 格式返回

def api_comment_json(comment_id):
    """ 以 JSON 格式返回留言数据的 API
    """
    comment = get_comment(comment_id)  # 获取 Comment 对象
    comment_dict = mapping_comment(comment)  # 映射到字典
    return json.dumps(comment_dict, indent=2)  # 以 JSON 格式返回

def main():
    # 获取用户数据的 JSON 并显示
    print "--- api_user_json ---"
    print api_user_json('bp12345')
    # 获取用户数据（详细）的 JSON 并显示
    print "--- api_user_detail_json ---"
    print api_user_detail_json('bp12345')
    # 获取留言数据的 JSON 并显示
    print "--- api_comment_json ---"
    print api_comment_json('cm54321')

if __name__ == '__main__':
    main()
```

在这段代码中，实现 API 功能的函数有 api_user_json、api_user_detail_json、api_comment_json。其执行结果如 LIST 15.11 所示。

LIST 15.11　执行结果

```
$ python mapping_model.py
--- api_user_json ---
{
  "user_id": "bp12345",
  "user_nickname": "tokibito"
}
--- api_user_detail_json ---
{
  "user_id": "bp12345",
  "user_nickname": "tokibito",
  "user_age": 26
}
--- api_comment_json ---
{
  "text": "Hello, world!",
  "user": {
    "user_id": "bp12345",
    "user_nickname": "tokibito"
  }
}
```

在 api_comment_json 的响应中，user 部分的数据结构要与 api_user_json 保持一致，因此使用了相同的映射函数。相对地，虽然 api_user_detail_json 与 api_user_json 的结构大致相同，但它们具有差异的部分使得它们用了不同的映射函数。

像上面这样，由于每个 API 之间都只存在细微的差异，使得映射函数成了一个俄罗斯套娃般的结构。随着这种函数增多，代码的可读性会越来越差。另外，因 API 的需求变更而导致函数传值参数增加时，需要一次性修正多个地方。

这些问题可以通过导入 bpmappers 来解决。

15.2.3　导入 bpmappers

bpmappers 能帮助我们将对象或字典的数据映射到其他字典上。bpmappers 通过 pip 命令进行安装，代码如 LIST 15.12 所示。本书使用的 bpmappers 版本是 0.8。

LIST 15.12　用 pip 命令安装 bpmappers

```
$ pip install bpmappers
```

bpmappers 主要由 Mapper 类和 Field 类构成。Mapper 类相当于映射函数，Field 类相当于映射字典的键值对。我们通过 Python shell 执行 bpmappers，做一个简单的映射（LIST 15.13）。

LIST 15.13　用 bpmappers 做映射

```
>>> from bpmappers import Mapper, RawField
>>> class SpamMapper(Mapper):
...     spam = RawField('foo')
...     egg = RawField('bar')
...
>>>
>>> SpamMapper(dict(foo=123, bar='abc')).as_dict()
{'egg': 'abc', 'spam': 123}
```

例子中定义了继承 Mapper 类的 SpamMapper 类，其属性包含 spam 和 egg 两个 RawField 对象。生成 SpamMapper 类的实例时，传值参数中指定了用做映射对象的字典。映射后的字典可以通过执行 Mapper 类的 as_dict 方法来获取。SpamMapper 类将 foo 键（或属性）的值映射到了 spam 键，将 bar 键（或属性）的值映射到了 egg 键。

接下来我们对前面那个返回用户数据和留言数据的 API（mapping_model.py）的源码作一下修改，对其导入 bpmappers。类和函数的重复部分在此省略。

LIST 15.14　bpmappers_mapping_model.py

```
# coding: utf-8
import json
from bpmappers import Mapper, RawField, DelegateField

class User(object):
    "省略"

class Comment(object):
    "省略"

def get_user(user_id):
    "省略"

def get_comment(comment_id):
    "省略"

class UserMapper(Mapper):
```

```
    """User 模型与 API 的映射
    """
    user_id = RawField('id')
    user_nickname = RawField('nickname')

class UserMapper2(UserMapper):
    """User 模型与 API 的映射 2
    """
    user_age = RawField('age')

class CommentMapper(Mapper):
    """Comment 模型与 API 的映射
    """
    user = DelegateField(UserMapper)
    text = RawField()

def api_user_json(user_id):
    """ 以 JSON 格式返回用户数据的 API
    """
    user = get_user(user_id)  # 获取 User 对象
    user_dict = UserMapper(user).as_dict()  # 映射到字典
    return json.dumps(user_dict, indent=2)  # 以 JSON 格式返回

def api_user_detail_json(user_id):
    """ 以 JSON 格式返回用户详细数据的 API
    """
    user = get_user(user_id)  # 获取 User 对象
    user_dict = UserMapper2(user).as_dict()  # 映射到字典
    return json.dumps(user_dict, indent=2)  # 以 JSON 格式返回

def api_comment_json(comment_id):
    """ 以 JSON 格式返回留言数据的 API
    """
    comment = get_comment(comment_id)  # 获取 Comment 对象
    comment_dict = CommentMapper(comment).as_dict()  # 映射到字典
    return json.dumps(comment_dict, indent=2)  # 以 JSON 格式返回

def main():
    " 省略 "

if __name__ == '__main__':
    main()
```

LIST 15.14 的执行结果没有变化。api_user_json 使用了 UserMapper。api_user_detail_json 使用的是继承 UserMapper 且添加了 age 映射的 UserMapper2 类。可以看到，bpmappers 的 Mapper 类能够利用继承的结构来添加不同的映射。另外，api_comment_json 的 user 部分与 UserMapper 的数据结构相同，所以我们直接通过 DelegateField 指定了 UserMapper。这种俄罗斯套娃式的映射结构同样可以用其他类来实现。

另外，列表内元素的套娃式映射可以用 ListDelegateField 来完成。LIST 15.15 中，我们通过 Python shell 执行了一个用 ListDelegateField 实现的映射。

LIST 15.15　用 ListDelegateField 实现的映射

```
>>> from bpmappers import Mapper, RawField, ListDelegateField
>>> class SpamMapper(Mapper):
...     spam = RawField('foo')
...
>>> class ListSpamMapper(Mapper):
...     spam_list = ListDelegateField(SpamMapper)
...
>>> ListSpamMapper({'spam_list': [{'foo': 123}, {'foo': 456}]}).as_dict()
{'spam_list': [{'spam': 123}, {'spam': 456}]}
```

ListDelegateField 中指定了继承 Mapper 类的 SpamMapper 类。ListDelegateField 可以以指定的类映射列表中的各个元素。通过上述例子我们可以看到，用 bpmappers 能够简化映射定义，同时方便映射的重复利用。

15.2.4　与 Django 联动

bpmappers 的一些功能可以为 Django 框架的模型对象映射提供辅助。使用 bpmappers.djangomodel.ModelMapper 可以轻松地根据 Django 的模型类生成用于映射的类。

下面我们用 ModelMapper 类来给简单的 Django 模型类作一个映射。请注意，这里我们不创建 Django 工程，所以需要在源码内初始化 Django（LIST 15.16）。

LIST 15.16　django_and_bpmappers.py

```
# coding: utf-8
# 初始化 Django
from django.conf import settings
settings.configure()

from django.db import models
from bpmappers.djangomodel import ModelMapper
```

```
class Person(models.Model):
    """ 表示人的数据模型
    """
    name = models.CharField(u' 名字 ', max_length=20)
    age = models.IntegerField(u' 年龄 ')

    class Meta:
        # 指定 app_label，防止应用名解析时出错
        app_label = ''

class PersonMapper(ModelMapper):
    """ 让 Person 模型映射到字典时需要用到的类
    """
    class Meta:
        model = Person

def main():
    # 生成 Person 对象
    person = Person(id=123, name=u'okano', age=26)
    # 映射到字典
    person_dict = PersonMapper(person).as_dict()
    # 输出到屏幕上
    print person_dict

if __name__ == '__main__':
    main()
```

为了让 Person 模型映射到字典，我们定义了一个继承 ModelMapper 类的 PersonMapper 类。ModelMapper 类内部定义了内部类 Meta，model 指定了 Person 模型。这样描述之后，ModelMapper 就会自动地根据 Person 模型拥有的字段生成映射。

在安装了 bpmappers 和 Django 的计算机上运行上述代码将得到如 LIST 15.17 所示的结果。

LIST 15.17　执行结果

```
$ python django_and_bpmappers.py
{'id': 123, 'name': u'okano', 'age': 26}
```

15.2.5　编写 JSON API

接下来我们在导入 bpmappers 的前提下实际编写一个返回 JSON 格式响应的 API。首先，我们以第 2 章中开发的留言板应用为例编写代码，实现在用户提交信息时以 JSON 格式返回响应。具体代码如下。

```
from flask import jsonify
from bpmappers import Mapper, RawField, ListDelegateField

class GreetingMapper(Mapper):
    name = RawField()
    comment = RawField()

class GreetingListMapper(Mapper):
    greeting_list = ListDelegateField(GreetingMapper)

@application.route('/api/')
def api_index():
    """ 留言
    """
    # 读取提交的数据
    greeting_list = load_data()
    result_dict = GreetingListMapper(
        {'greeting_list': greeting_list}).as_dict()
    # 以 JSON 格式返回响应
    return jsonify(**result_dict)
```

将这段代码添加到 guestbook.py 的 `if __name__` …之前。JSON 的响应会以 `greeting_list` 为键，通过数组的形式返回各次提交的姓名以及留言内容。要返回的数据通过已有的 load_data 函数获取，然后以 GreetingListMapper 类进行映射。GreetingListMapper 类使用了 ListDelegateField 类，从而实现以 GreetingMapper 类对列表内的值进行映射。

接下来保存修改，执行源码并启动服务器。在添加几条数据之后访问 `http://127.0.0.1:5000/api/`，我们会得到 JSON 格式的响应。下面是用 urllib 访问时的例子。

```
$ python -m urllib http://127.0.0.1:5000/api/
{
  "greeting_list": [
    {
      "comment": "\u65e5\u672c\u8a9e\u306e\u6587\u5b57\u5217",
      "name": "tokibito"
    },
    {
      "comment": "Hello, world!",
      "name": "tokibito"
    }
  ]
}
```

导入 bpmappers 能提高映射的重复利用率，还能让我们在需求变更时更灵活地加以应对，因此即便是很简单的 API，也建议用 bpmappers 来实现。

15.3　图像处理

Python 的图像处理通常用 Pillow（Python Imaging Library（Fork））来进行。Pillow 由 PIL（Python Imaging Library）的分支工程开发而来。由于 PIL 已经停止开发及维护，所以如今 Pillow 成为了主流。它支持 JPEG、PNG、GIF、BMP 等多种图像格式。本书使用的是 Pillow 的 2.6.1 版本。

> Pillow
> `http://pillow.readthedocs.org/`

15.3.1　安装 Pillow

Pillow 与多种处理图像数据的程序库存在依赖关系，因此安装时需要多加注意。目前 Pillow 在 PyPI 上提供了面向 Windows 和 OS X 的 wheel 包。在 Windows、OS X 上安装（包括用 pip 命令安装）时不需要进行编译。如果使用的是其他平台，那么由于需要从 sdist 进行 C 扩展的编译，所以必须准备编译器和各种图像处理库。

◉ 有 wheel 可用的平台

如果是 OS X 和 Windows，只需像 LIST 15.18 这样使用 `pip install` 安装 wheel 包即可。

LIST 15.18　在 OS X 上安装 Pillow

```
$ pip install pillow==2.6.1
Downloading/unpacking pillow==2.6.1
  Downloading Pillow-2.6.1-cp27-none-macosx_10_6_intel.macosx_10_9_intel.
macosx_10_9_x86_64.whl (2.8MB): 2.8MB downloaded
Installing collected packages: pillow
Successfully installed pillow
Cleaning up...
```

◉ 从源码构建

接下来准备进行 Pillow 编译时所需的库。下面以 Ubuntu 14.04 为例进行学习。

首先，因为需要编译 C 扩展，所以需要一些基本的开发工具。我们先来确认一下 1.1 节中

的安装（LIST 15.19）。

LIST 15.19　检查设置以便进行 python 的 C 扩展编译

```
$ pkg-config python-2.7 --libs --cflags
-I/usr/include/python2.7 -I/usr/include/x86_64-linux-gnu/python2.7  -lpython2.7
```

另外，图像格式和字体等的支持需要用到下述程序库。

支持对象	库
JPEG	libjpeg-dev
OpenJPEG	libopenjpeg-dev
PNG	zlib1g-dev
TIFF	libtiff5-dev
webp	libwebp-dev
字体	libfreetype6-dev
色彩管理	liblcms2-dev

执行 LIST 15.20 中的命令，统一安装 Pillow 需要的程序包。

LIST 15.20　安装 Pillow 需要的程序包

```
$ sudo apt-get install libjpeg-dev libopenjpeg-dev zlib1g-dev libtiff5-dev
libfreetype6-dev libwebp-dev liblcms2-dev
```

现在所需工具和库已经齐全，可以用 pip 进行安装了（LIST 15.21）。

LIST 15.21　用 pip 命令安装 Pillow

```
$ pip install pillow==2.6.1
```

安装时会显示支持的图像格式等，我们可以借此查看想要的功能是否已经生效。LIST 15.22 是除 TKINTER 以外的所有功能均生效的例子。

LIST 15.22　查看支持的功能

```
running build_ext
--------------------------------------------------------------------
PIL SETUP SUMMARY
--------------------------------------------------------------------
version      Pillow 2.6.1
platform     linux2 2.7.6 (default, Mar 22 2014, 22:59:56)
             [GCC 4.8.2]
--------------------------------------------------------------------
*** TKINTER support not available
```

```
--- JPEG support available
--- OPENJPEG (JPEG2000) support available (2.1.3)
--- ZLIB (PNG/ZIP) support available
--- LIBTIFF support available
--- FREETYPE2 support available
--- LITTLECMS2 support available
--- WEBP support available
--- WEBPMUX support available
--------------------------------------------------------------------
To check the build, run the selftest.py script.
```

NOTE

Pillow 2.6.1 无法识别 Ubuntu 14.04 上安装的 libopenjpeg-dev。今后的版本中应该会修复这个问题。

如果应用不涉及 Tkinter 模块的图像，可以不用管 TKINTER 的支持问题。另外，安装 Python 时，如果 Tkinter 模块并未生效，同样无法支持 TKINTER。

15.3.2　图像格式转换

图像文件的格式转换通过在 Image 类的 save 方法的传值参数中指定格式并保存来完成。下面，我们打开当前目录下名为 python.gif 的图像文件，将其转换为 JPEG 格式，并保存在 python_convert.jpg 文件中。具体代码如下。

```
# coding: utf-8
from PIL import Image

def main():
    # 打开文件获取 Image 对象
    image = Image.open('python.gif')
    # 模式转换为 RGB
    image_rgb = image.convert('RGB')
    # 图像保存至文件
    image_rgb.save('python_convert.jpg', 'jpeg')

if __name__ == '__main__':
    main()
```

可以看到，程序在读取完文件之后将图像模式转为了 RGB。

在 GIF 以及不足 256 色的 PNG、BMP 等格式中，颜色信息都保存在调色板数据块里。这

类文件用 Pillow 打开时分为 P 模式（调色板模式）和 1 模式（单色模式）。另外，JPEG 文件有时还会是 CMYK 模式。当模式不支持 save 方法指定的格式时，程序会报错，所以要先用 convert 方法进行模式转换。

15.3.3　改变图像尺寸

如果想改变图像尺寸，可以使用 Image 类的 thumbnail 方法或 resize 方法。下面，我们打开当前目录下名为 python.jpg 的图像文件，将其长宽缩小一半后保存为 python_thumbnail.jpg，代码如 LIST 15.23 所示。

LIST 15.23　pil_thumbnail.py

```
# coding: utf-8
from PIL import Image

def main():
    # 打开文件获取 Image 对象
    image = Image.open('python.jpg')
    # 计算图像长宽的一半
    half_size = (image.size[0] / 2, image.size[1] / 2)
    # 图像大小降为一半
    image.thumbnail(half_size, Image.ANTIALIAS)
    # 图像保存至文件
    image.save('python_thumbnail.jpg')

if __name__ == '__main__':
    main()
```

Image 类的对象能够通过 size 属性以元组的形式获取图像的长和宽。

thumbanil 方法的第一个传值参数指定了图像长和宽的元组，第二个传值参数指定了滤镜 Image.ANTIALIAS。滤镜有 NEAREST、BILINER、BICUBIC、ANTIALIAS4 种可供选择，其中使用 ANTIALIAS 修改尺寸后的图像品质最高（损失最小）。

在执行 thumbnail 方法之后，会直接修改对象自身的图像大小。但是，这个方法只能用于长宽比例不变的修改。变更长宽比例时需要使用 resize 方法。下面，我们打开当前目录下名为 python.jpg 的图像文件，将其长度放大为 2 倍后保存为 python_resize.jpg，具体代码如 LIST 15.24 所示。

LIST 15.24　pil_resize.py

```
# coding: utf-8
from PIL import Image
```

```
def main():
    # 打开文件获取 Image 对象
    image = Image.open('python.jpg')
    # 计算图像长度的 2 倍
    double_size = (image.size[0], image.size[1] * 2)
    # 图像大小增加至 2 倍
    image_resized = image.resize(double_size, Image.ANTIALIAS)
    # 图像保存至文件
    image_resized.save('python_resize.jpg')

if __name__ == '__main__':
    main()
```

与 thumbnail 方法不同，resize 方法的返回值是修改尺寸后的 Image 类的对象。它同 thumbnail 一样，可以指定滤镜。图 15.1 和图 15.2 分别是修改尺寸之前的图像与执行完 LIST 15.24 所示的代码之后的图像。



图 15.1　修改尺寸之前的图像（python.jpg）



图 15.2　修改尺寸之后的图像（python_resize.jpg）

15.3.4　剪裁图像

Image 类的 crop 方法能够以长方形剪裁图像。下面，我们打开当前目录下名为 python.jpg 的图像文件，按照图像的宽度从正中间剪裁一个正方形并保存为 python_crop.jpg。

LIST 15.25　pil_crop.py

```
# coding: utf-8
from PIL import Image

def main():
    # 打开文件获取 Image 对象
    image = Image.open('python.jpg')
    # 根据短边长度求中央正方形的坐标
    if image.size[0] < image.size[1]:
        # 横边较短时(瘦高的图像)
        crop_rect = (
            0,
            (image.size[1] - image.size[0]) / 2,
            image.size[0],
            (image.size[1] - image.size[0]) / 2 + image.size[0])
    else:
        # 竖边较短时(矮胖的图像)
        crop_rect = (
            (image.size[0] - image.size[1]) / 2,
            0,
            (image.size[0] - image.size[1]) / 2 + image.size[1],
            image.size[1])
    # 剪裁
    image_croped = image.crop(crop_rect)
    # 图像保存至文件
    image_croped.save('python_crop.jpg')

if __name__ == '__main__':
    main()
```

crop 方法的传值参数是包含 4 个值的元组（Tuple），这 4 个值代表长方形剪裁区域的左上角坐标和右下角坐标。crop 的返回值为存有剪裁后图像的 Image 类对象。执行 LIST 15.25 中的代码后会得到如图 15.3 所示的结果。



图 15.3　剪裁后的图像（python_crop.jpg）

15.3.5　对图像进行滤镜处理

进行滤镜处理必须获取像素值。像素值可以用 Image 类的 getdata 方法或 getpixel 方法来获取。获取的像素值为包含 R（红）、G（绿）、B（蓝）3 个值的元组，3 个值的范围均为 0 ~ 255。下面，我们打开当前目录下名为 python.jpg 的图像文件，将所有像素反色并保存为 python_filter.jpg。

LIST 15.26　pil_filter.py

```
# coding: utf-8
from PIL import Image

def main():
    # 打开文件获取 Image 对象
    image = Image.open('python.jpg')
    buffer = []
    # 循环逐一获取图像的像素
    for pixel in image.getdata():
        # 将像素反色并存入缓冲区
        buffer.append((
            255 - pixel[0],
            255 - pixel[1],
            255 - pixel[2]))
    # 用缓冲区内的像素覆盖原有数据
    image.putdata(buffer)
    # 图像保存至文件
    image.save('python_filter.jpg')

if __name__ == '__main__':
    main()
```

getdata 方法能够返回一个迭代器，用于逐一访问图像的每一组像素值。在上例中，我们逐一取出了每个像素的像素值并进行反色（255 减去色值）。等所有像素值处理完毕之后，用 putdata 方法替换了 Image 类的对象的像素。LIST 15.26 的执行结果如图 15.4。



图 15.4　反色后的图像 (python_filter.jpg)

如果要获取指定坐标的像素值，可以用 Image 类的 getpixel 方法。下面，我们打开当前目录下名为 python.jpg 的图像文件，将右上角的像素反色并保存为 python_pixel.jpg，具体代码如 LIST 15.27 所示。

LIST 15.27　pil_pixel.py

```
# coding: utf-8
from PIL import Image

def main():
    # 打开文件获取 Image 对象
    image = Image.open('python.jpg')
    # 右上角的位置
    point = (image.size[0] - 1, 0)
    # 获取像素值
    pixel = image.getpixel(point)
    # 改写为反色的像素值
    image.putpixel(point, (
        255 - pixel[0],
        255 - pixel[1],
        255 - pixel[2]))
    # 图像保存至文件
    image.save('python_pixel.jpg')

if __name__ == '__main__':
    main()
```

getpixel 方法的传值参数为含有横纵坐标 (起点为 0) 两个值的元组。改写指定位置像素值时使用了 putpixel 方法。这些方法的方便之处在于能够指定坐标，但是速度太慢，因此一旦需要大量使用，它们的效率并不见得比 getdata、putdata 等方法更高。

15.4　数据加密

在数据传输过程中，为防止数据被第三方获取或篡改，需要对数据进行加密。这里我们学习一下如何通过 PyCrypto 进行通用加密系统和公钥加密系统的加密及解密。

> PyCrypto
> `https://pypi.python.org/pypi/pycrypto`

15.4.1　安装 PyCrypto

如 LIST 15.28 所示，PyCrypto 的安装可以通过 pip 命令进行。由于它包含 C 语言编写的模块，所以安装时与 Pillow 一样需要 gcc 等编译器。各位可以事先用 apt 安装 python-dev 包和 build-essential 包。本书使用的是 PyCrypto 的 2.6.1 版本。

LIST 15.28　用 pip 命令安装 PyCrypto

```
$ pip install pycrypto==2.6.1
```

15.4.2　通用加密系统的加密及解密

通用加密系统在加密和解密时使用同一套密钥。AES、DES 等都属于通用加密算法。AES 的密码长度和块长都高于 DES，因此安全性较高。本书使用的就是 AES。

> **NOTE**
> DES（Data Encryption Standard）是 1977 年被美国标准化的加密系统。
> AES（Advanced Encryption Standard）是 2011 年被美国标准化的加密系统。

以 AES 加密、解密时需要用到 PyCrypto 的 Crypto.Cipher.AES 类。下面我们用 PyCrypto 实现 AES 加密及解密，并将结果输出到屏幕上（LIST 15.29）。

LIST 15.29　aes_encrypt.py

```
# coding: utf-8
from Crypto.Cipher import AES

KEY = 'testtesttesttest'  # 加密和解密时使用的通用密钥
DATA = '0123456789123456'  # 数据长度为 16 的倍数
```

```
def main():
    aes = AES.new(KEY)  # 生成 AES 类的实例
    encrypt_data = aes.encrypt(DATA)  # 加密
    print repr(encrypt_data)  # 输出至屏幕
    decrypt_data = aes.decrypt(encrypt_data)  # 解密
    print repr(decrypt_data)  # 输出至屏幕

if __name__ == '__main__':
    main()
```

给 AES.new 函数指定用作密钥的字符串，生成 AES 对象。密钥可以是长度为 16、24、32 字符的任意字符串。数据通过 AES 对象的 encrypt 方法加密，通过 decrypt 方法解密。上述代码段的执行结果如 LIST 15.30 所示。

LIST 15.30　执行结果

```
$ python aes.py
'\xf7\xf1\x0ccS\xef\x02\xe10\xf2\xe1\xd2\x80{\xcf\xfa'
'0123456789123456'
```

执行结果的第一条输出是加密状态的数据，第二条是将加密的二进制串解密后还原的数据。

15.4.3　公钥加密系统（RSA）的加密与解密

公钥加密系统在加密和解密时分别使用不同的密钥。RSA 等就是公钥加密算法。

> **NOTE**
>
> RSA 是 Ron Rivest、Adi Shamir、Len Adleman 于 1977 年发明的加密算法。

◉ RSA 私钥和公钥的生成

在公钥加密系统中，加密使用公钥（Public key），解密使用私钥（Private key）。这两种密钥都需要通过算法生成。公钥和私钥的密钥对可以通过 ssh-keygen 命令或 openssl 命令来创建，不过我们这里要学习的是用 PyCrypto 生成密钥的方法。下面，我们用 PyCrypto 生成 RSA 的密钥对，以 PEM 格式（RFC1421）输出到屏幕上，具体代码如下。

LIST 15.31　rsa_generate_keypair.py

```
# coding: utf-8
from Crypto.PublicKey import RSA
from Crypto import Random
```

```
INPUT_SIZE = 1024

def main():
    random_func = Random.new().read  # 产生随机数的函数
    key_pair = RSA.generate(INPUT_SIZE, random_func)  # 生成密钥对
    private_pem = key_pair.exportKey()  # 获取 PEM 格式的私钥
    public_pem = key_pair.publickey().exportKey()  # 获取 PEM 格式的公钥
    print private_pem  # 输出至屏幕
    print public_pem  # 输出至屏幕

if __name__ == '__main__':
    main()
```

LIST 15.31 用 RSA.generate 函数生成了 RSA 密钥对的对象，用 exportKey 方法获取了用作私钥的 PEM 格式的字符串。公钥则是先通过 publickey 方法获取对象，然后再用 exportKey 方法获取的。上述代码的执行结果如 LIST 15.32 所示。

LIST 15.32　执行结果

```
$ python rsa_generate_keypair.py
-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDMYS234o1C1Z2fbeZazcUnEfspBcs06hSmvDji+Jm5Gk6tvIHl
IFFu1aCD8kBbjf2ivzmG8Dgtcn6jnLjXe3EB0H1vh70TUsvi0ZjxZsmbv6fJmJrQ
zJvW1Wi3wnoBeVYQk6ha8rbfY35wErxxdTWeWm1nSBwaFfnRFYnrkqVGlQIDAQAB
AoGBAKJZ39Ne6A/bWOa4inA/XQl4QyeHLrDN8bGxew7xpEtiFnX0dMrqLUX59RRb
b7xKwtxxQuVqFXYkqWyWpk6mBFGcRH1yH888Cgu+mSbsKvMAGOW/oTl7XLV8hc4T
m0iT/gEUsCHFcE6mstkUIEMlZCWmnuoijprDbehh1OSEZPQBAkEA1IFgXqMGIC/x
CYwrizFgJVAa/o4IF183CocfqPaYlotKCeNovnPXeSCmAX1d0GhCHKBIQmkmL7YU
TZ1DxiWL1QJBAPY2CWyA26GKGu1WzURJa7guizaqGJpghF30U5VdvdKmetYU2gXA
rhHQ9LxdjG09L9BWSxg5Y1Zl02b8f2Qf78ECQQCNr3VBpBCBhXWAmCSwOcuRFUfq
UWizrJhWPKGvVjuGpHhI/4bm9PXFnS8R7zSNr/XkgDmtjc4YIZ6H4UM+6enBAkBi
yC9jvxdfan9/NdJJUYPMc7AbEIeqeIri/0IBrYiZWX3zIo6OvE2ajFGEuau7sE7c
saKTZ4L5iQUWTrv1ufKBAkEAis4KsI4Inxz01ZPRcmPlUVKULvVqyquqsfKP+NFG
PTurYiXOc2kXPbBNxyhTDQ6Dw3OB0GhARHSGiuhQQicA2w==
-----END RSA PRIVATE KEY-----
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDMYS234o1C1Z2fbeZazcUnEfsp
Bcs06hSmvDji+Jm5Gk6tvIHlIFFu1aCD8kBbjf2ivzmG8Dgtcn6jnLjXe3EB0H1v
h70TUsvi0ZjxZsmbv6fJmJrQzJvW1Wi3wnoBeVYQk6ha8rbfY35wErxxdTWeWm1n
SBwaFfnRFYnrkqVGlQIDAQAB
-----END PUBLIC KEY-----
```

在输出的字符串中，从 -----BEGIN RSA PRIVATE KEY----- 到 -----END RSA PRIVATE KEY----- 的部分为私钥，从 -----BEGIN PUBLIC KEY----- 到 -----END PUBLIC KEY-----

的部分为公钥。加密解密时就是使用这一对密钥。

◉ 用公钥加密

加密需要使用公钥。PyCrypto 可以使用我们输入的 PEM 格式的公钥字符串。下面，我们将字符串 `Hello, world!` 加密并输出到屏幕上（LIST 15.33）。

LIST 15.33　rsa_encrypt.py

```
# coding: utf-8
from Crypto.PublicKey import RSA
from Crypto import Random

DATA = 'Hello, world!'
PUBLIC_KEY_PEM = """-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDMYS234o1C1Z2fbeZazcUnEfsp
Bcs06hSmvDji+Jm5Gk6tvIHlIFFu1aCD8kBbjf2ivzmG8Dgtcn6jnLjXe3EB0H1v
h70TUsvi0ZjxZsmbv6fJmJrQzJvW1Wi3wnoBeVYQk6ha8rbfY35wErxxdTWeWm1n
SBwaFfnRFYnrkqVGlQIDAQAB
-----END PUBLIC KEY-----"""

def main():
    random_func = Random.new().read  # 产生随机数的函数
    public_key = RSA.importKey(PUBLIC_KEY_PEM)  # 输入 PEM 格式的公钥
    encrypted = public_key.encrypt(DATA, random_func)  # 加密数据
    print encrypted  # 输出至屏幕

if __name__ == '__main__':
    main()
```

这段代码用 RSA.importKey 函数输入公钥并获取了 RSA 对象，然后用 encrypt 方法进行加密。encrypt 方法的传值参数处指定了需要加密的数据以及产生随机数的函数。上述代码的执行结果如 LIST 15.34 所示。

LIST 15.34　执行结果

```
$ python rsa_encrypt.py
('\x05 a\\\xb9U\x88/El\x1a\x02\xe6\xb4\xede\xf2\xe6\xe3\xa6&~\x9e\x180[K%i\x02k
\xdd\xd5%\xfd\x1a\xc6\xd7\xc4\xa8\xcf\x86\x07\xdck\x7f\xb4\xb5_,I\x80\xe9\x83\
x00*q\xce\xacA\x9a\xe3$]\xe5*\x9e\x91F\xd2\xe3P\xb8+\xa6\xc1R\xde\xf2G\xf1\x185\
xcd\x8f\x82\x1a\xa4c\xf5\x9c\xd8\xe0\xd1g \xfdw\xa0\xe6\xca\xf7\x9f\xde\xbf(\xa
2\xd5\xdb\xd5}\xe5\xaf\x99\xf9\x90\x1cx\n\xe8\xda\x14\x9cJ\xd7\xe4\x96S',)
```

◉ 用私钥解密

解密需要使用私钥。与加密时一样，这里也要输入 PEM 格式的字符串。下面，我们把前面加密的数据解密并显示在屏幕上，代码如 LIST 15.35 所示。

LIST 15.35　rsa_decrypt.py

```
# coding: utf-8
from Crypto.PublicKey import RSA

DATA = ('\x05 a\\\xb9U\x88/El\x1a\x02\xe6\xb4\xede\xf2\xe6\xe3\xa6&~\x9e\x180[K
%i\ x02k\xdd\xd5%\xfd\x1a\xc6\xd7\xc4\xa8\xcf\x86\x07\xdck\x7f\xb4\xb5_,I\x80\xe
9\x83\x00*q\xce\xacA\x9a\xe3$]\xe5*\x9e\x91F\xd2\xe3P\xb8+\xa6\xc1R\xde\xf2G\xf1
\x185\xcd\x8f\x82\x1a\xa4c\xf5\x9c\xd8\xe0\xd1g \xfdw\xa0\xe6\xca\xf7\x9f\xdexbf
(\xa2\xd5\xdb\xd5}\xe5\xaf\x99\xf9\x90\x1cx\n\xe8\xda\x14\x9cJ\xd7\xe4\x96S',)
PRIVATE_KEY_PEM = """-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDMYS234o1C1Z2fbeZazcUnEfspBcs06hSmvDji+Jm5Gk6tvIHl
IFFu1aCD8kBbjf2ivzmG8Dgtcn6jnLjXe3EB0H1vh70TUsvi0ZjxZsmbv6fJmJrQ
zJvW1Wi3wnoBeVYQk6ha8rbfY35wErxxdTWeWm1nSBwaFfnRFYnrkqVGlQIDAQAB
AoGBAKJZ39Ne6A/bWOa4inA/XQl4QyeHLrDN8bGxew7xpEtiFnX0dMrqLUX59RRb
b7xKwtxxQuVqFXYkqWyWpk6mBFGcRH1yH888Cgu+mSbsKvMAGOW/oTl7XLV8hc4T
m0iT/gEUsCHFcE6mstkUIEMlZCWmnuoijprDbehh1OSEZPQBAkEA1IFgXqMGIC/x
CYwrizFgJVAa/o4IF183CocfqPaYlotKCeNovnPXeSCmAX1d0GhCHKBIQmkmL7YU
TZ1DxiWL1QJBAPY2CWyA26GKGu1WzURJa7guizaqGJpghF30U5VdvdKmetYU2gXA
rhHQ9LxdjG09L9BWSxg5Y1Zl02b8f2Qf78ECQQCNr3VBpBCBhXWAmCSwOcuRFUfq
UWizrJhWPKGvVjuGpHhI/4bm9PXFnS8R7zSNr/XkgDmtjc4YIZ6H4UM+6enBAkBi
yC9jvxdfan9/NdJJUYPMc7AbEIeqeIri/0IBrYiZWX3zIo6OvE2ajFGEuau7sE7c
saKTZ4L5iQUWTrv1ufKBAkEAis4KsI4Inxz01ZPRcmPlUVKULvVqyquqsfKP+NFG
PTurYiXOc2kXPbBNxyhTDQ6Dw3OB0GhARHSGiuhQQicA2w==
-----END RSA PRIVATE KEY-----"""

def main():
    # 输入 PEM 格式的私钥
    private_key = RSA.importKey(PRIVATE_KEY_PEM)
    # 解密数据
    decrypted = private_key.decrypt(DATA)
    # 输出至屏幕
    print decrypted

if __name__ == '__main__':
    main()
```

与加密时一样，解密也是通过 RSA.importKey 函数输入私钥并获取 RSA 对象。接下来，我们在 decrypt 方法的传值参数中指定已加密的数据进行解密。上述代码的执行结果如 LIST 15.36 所示。

LIST 15.36　执行结果

```
$ python rsa_decrypt.py
Hello, world!
```

NOTE

为了便于理解，上述例子都是直接在代码中描述公钥和私钥。实际使用时可以从密钥文件中读取密钥数据，从而提高效率。

15.5　使用 Twitter 的 API

随着 Twitter 在全世界推广，系统与 Twitter 联动的需求越来越常见。现在有不少封装了 Twitter API 的 Python 模块，这里我们以 tweepy 为例学习如何使用 tweepy 模块。tweepy 几乎涵盖了所有 Twitter API，而且能相对灵活地应对 Twitter API 自身的规格变更。下面我们用 Flask 和 tweepy 来简单做一个基于 Web 的时间轴视图。

tweepy
`http://www.tweepy.org/`

15.5.1　导入 tweepy

如 LIST 15.37 所示，通过 pip 命令安装 tweepy。本书使用了 tweepy 的 3.1.0 版本。

LIST 15.37　用 pip 命令安装 tweepy

```
$ pip install tweepy
```

15.5.2　添加应用与获取用户密钥

开发使用 Tiwtter 的 API 的应用时，需要将应用添加到 Twitter Application Management。下面我们来实际操作一下。如图 15.5 所示，Twitter Application Management 可以用 Twitter 账户登录。没有账户时需要新注册一个。登录后会进入 Twitter Apps 页面，该页上显示了所有已添加的应用。

Twitter Application Management
`http://apps.twitter.com/`

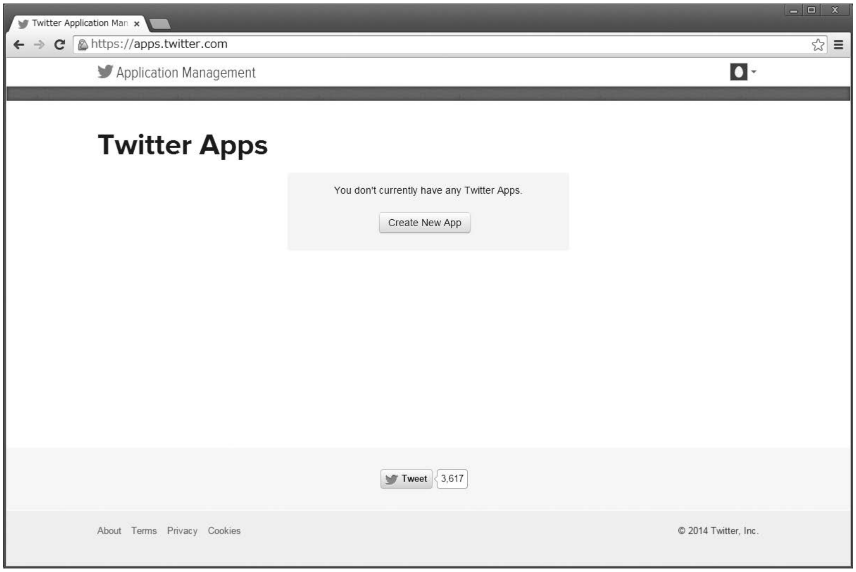


图 15.5　已添加应用的一览页面

添加新应用时，需要点击 Create New App 按钮进入添加页面，然后在该页面输入必填事项，如图 15.6 所示。

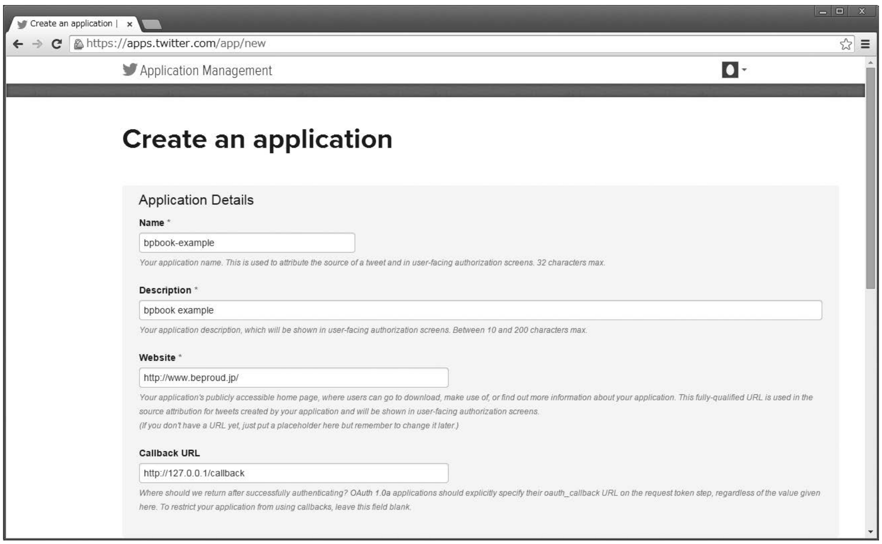


图 15.6　用于添加应用的页面

本书所用例子的输入如下。

Name（应用名）	bpbook-example
Description（应用的说明）	bpbook example
WebSite（应用的 Web 站点）	http://www.beproud.jp/
Callback URL（OAuth 认证后的回调 URL）	http://127.0.0.1:5000/callback
Yes, I agree（同意使用条款）	勾选后同意使用条款

系统上比较重要的只有 Callback URL 的值。这里指定的是通过 Twitter 站点认证后重定向访问的应用的 URL。我们需要在之后开发的应用中实现这个 URL 的处理。输入所有项目后点击 Create your Twitter application，如果没有问题，系统会提示添加完成，并显示已添加应用的详细信息，如图 15.7 所示。

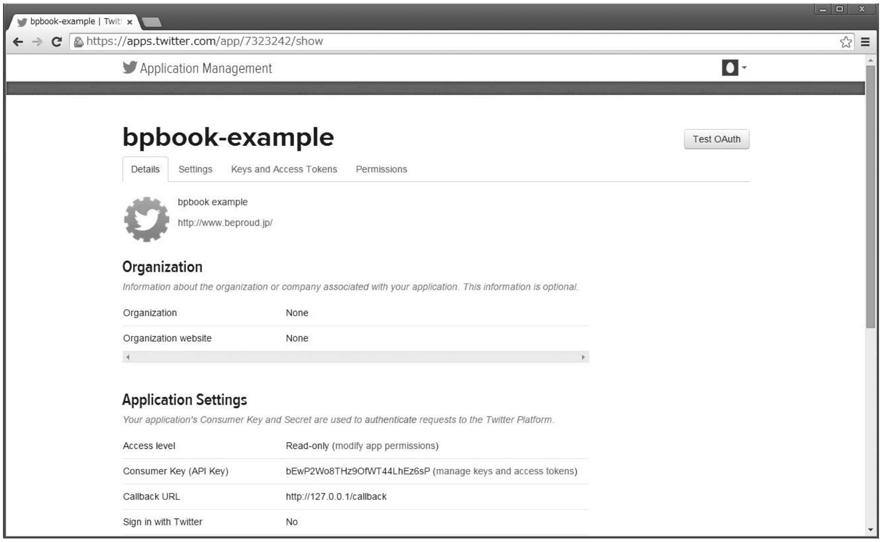


图 15.7　已添加应用的详细信息（添加完成时）

详细信息页面的 Keys and Access Tokens 标签页的 Application Settings 部分显示了 Consumer Key（用户密钥）和 Consumer Secret（用户机密）字符串。使用 Twitter 的 API 时需要用到这两个字符串以及下面即将学习的访问令牌和访问令牌机密。

Access Level 表示应用可以对用户数据做哪些操作，有 Read-only（只读）、Read and Write（可读写）、Read,Write and Access direct messages（可读写及访问私信）3 种操作可供选择。我们这里开发的时间轴视图只有读取操作，因此用 Read-only 就足够了。

15.5.3　获取访问令牌

OAuth 的访问令牌是为每一个使用应用的用户分别配发的值。可以通过已添加应用的详细信息页面为已添加应用的用户手动配发访问令牌。点击 Keys and Access Tokens 标签页下部的 Create my Access token 即可完成访问令牌的配发。点击 Regenerate My Access Token and Token Secret 可以作废已有令牌并重新配发新的。访问令牌以图 15.8 所示的字符串形式给出。我们将在应用中用到 Access Token（访问令牌）和 Access Token Secret（访问令牌机密）。

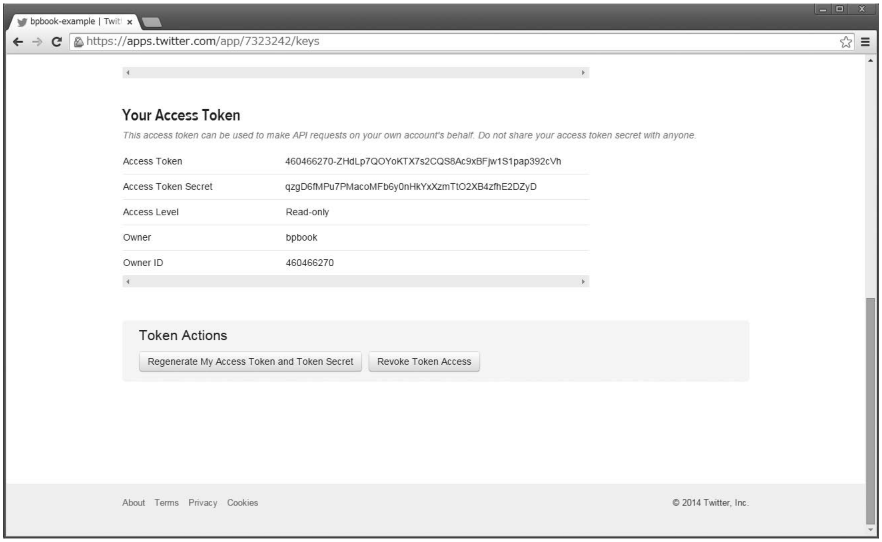


图 15.8　获取自己的访问令牌

如果应用只使用自己的 Twitter 账户，那么我们开发应用时只考虑上面获取的访问令牌即可。如果应用需要任意多个用户使用不同的 Twitter 账户，则需要使用基于 Web 的 Authorize URL 方式认证，从应用端获取访问令牌。我们即将开发的时间轴视图属于后者。

15.5.4　调用 Twitter API

用 tweepy 调用 Twitter API 时，需要用到 OAuthHandler 类和 API 类。下面，我们使用前面获取的用户密钥和访问令牌，获取 Twitter 主页上显示的时间轴，代码如 LIST 15.38 所示。

LIST 15.38　tweepy_hello.py

```
# coding: utf-8
from tweepy import OAuthHandler, API
```

```
# 用户密钥（本应用的）
CONSUMER_KEY = 'Consumer key'
CONSUMER_SECRET = 'Consumer secret'
# 访问令牌（bpbook 用户的）
ACCESS_TOKEN = 'Access token'
ACCESS_TOKEN_SECRET = 'Access token secret'

def main():
    # 生成 OAuth 的 handler
    handler = OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
    # 给 handler 设置访问令牌
    handler.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
    # 生成 API 实例
    api = API(handler)
    # 用 API 获取时间轴
    statuses = api.home_timeline()
    for status in statuses:
        # 将获取的状态的前 15 个字符输出到页面上
        print "%s: %s" % (status.user.screen_name, status.text[:15])

if __name__ == '__main__':
    main()
```

请各位自行将上述代码中的 CONSUMER_KEY、CONSUMER_SECRET、ACCESS_TOKEN、ACCESS_TOKEN_SECRET 部分替换为前面获取的值。先在 OAuthHandler 类的构造函数中指定用户密钥和用户机密，然后用 OAuthHandler 的 set_access_token 方法给 handler 设置对应的访问令牌。在 API 类的构造函数中指定 handler 对象，让我们能够使用 API 调用。没有设置访问令牌的 handler 无法使用绝大多数的 API 调用。home_timeline 方法最多可以返回 20 条首页的时间轴信息。

上述代码的执行结果如 LIST 15.39 所示。

LIST 15.39 执行结果

```
$ python tweepy_hello.py
bpbook: 测试
liblar_jp:【有关添加功能的通知】书籍相关
connpass_jp:【有关功能修改的通知】多天以来
liblar_jp:【有关维护工作已全面结束的通知】
beproud_jp: BePROUD 公司现在就 Web
（以下省略）
```

发生 TweepError: HTTP Error 401: Unauthorized 错误时，需要查看用户密钥和访问令牌是否

有误。另外，执行环境的系统时间与实际时间不一致也会导致这个错误。

除 home_timeline 之外，还有许多方法封装了 API。详细内容可以查看 tweepy 文档的 API Reference 来了解。

tweepy API Reference
　http://tweepy.readthedocs.org/en/v3.0.0/api.html

在这个例子中，我们是直接在代码中描述访问令牌（ACCESS_TOKEN、ACCESS_TOKEN_SECRET）的。对于只通过单一用户获取时间轴信息或进行发言的应用（比如 TwitterBot 等）而言，由于不需要用于认证的 UI，所以使用这种方法不会有问题。

15.5.5　编写用 Twitter 认证的系统

下面我们编写一个用 Twitter 账户登录（认证）并根据各 Twitter 账户的权限使用 API 的系统。编写这个系统中需要用到 Twitter 的认证页面，还要获取各个账户的访问令牌。从登录的起始处理到使用 API 的处理流程如下所示。

① 用户通过 Web 浏览器访问应用的登录 URL
② 应用从 Twitter 获取请求令牌
③ 应用将获取到的请求令牌保存在会话中
④ 应用向 Twitter 的认证 URL 返回带有令牌属性的重定向响应
⑤ 用户通过 Web 浏览器在 Twitter 的认证页面对应用授权
⑥ 用户根据 Twitter 端的重定向响应，通过 Web 浏览器访问应用的回调 URL（带有令牌属性）
⑦ 应用从会话中还原请求令牌
⑧ 应用从 Twitter 获取用户的访问令牌
⑨ 应用依据访问令牌使用 Twitter 的 API

使用 tweepy 可以轻松完成①的生成带属性的 URL、②的获取令牌、⑨的依据访问令牌使用 Twitter API。下面我们看一下 web 应用，它使用 Flask 和 tweepy 对 Twitter 账户进行认证，然后获取时间轴信息并使之显示在页面上，代码如 LIST 15.40 所示。

LIST 15.40　tweepy_auth.py

```
# coding: utf-8
from flask import Flask, render_template, session, request, redirect
```

```
from tweepy import OAuthHandler, API

application = Flask(__name__)
# 设置用于会话的密钥
application.secret_key = 'my secret key'

# 用户密钥
CONSUMER_KEY = 'Consumer key'
CONSUMER_SECRET = 'Consumer secret'

def get_access_token():
    """ 从会话获取访问令牌的函数
    """
    return session.get('access_token')

def set_access_token(access_token):
    """ 在会话中保存访问令牌的函数
    """
    session['access_token'] = access_token

def get_request_token(key):
    """ 从会话获取请求令牌的函数
    """
    return session.get(key)

def set_request_token(key, token):
    """ 在会话中保存请求令牌的函数
    """
    session[key] = token

def get_oauth_handler():
    """ 返回 Tweepy 的 OAuth handler 的函数
    """
    return OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)

def get_api(access_token):
    """ 指定访问令牌返回 API 实例的函数
    """
    handler = get_oauth_handler()
    handler.set_access_token(access_token[0], access_token[1])
    api = API(handler)
    return api
```

```
def set_user(user):
    """ 在会话中保存用户信息的函数
    """
    # 以可 JSON 序列化的字典对象格式保存在会话中
    session['user'] = {'screen_name': user.screen_name}

def get_user():
    """ 从会话获取用户信息的函数
    """
    return session.get('user')

def is_login():
    """ 返回是否为登录状态
    """
    return not not get_access_token()

def clear_session():
    """ 删除会话的值
    """
    session.clear()

@application.route('/')
def index():
    """ 首页
    使用模板显示页面
    """
    # 根据模板显示页面
    return render_template('index.html', is_login=is_login(), user=get_user())

@application.route('/login')
def login():
    """ 登录 URL
    开始 Twitter 认证
    """
    handler = get_oauth_handler()
    # 获取认证 URL
    auth_url = handler.get_authorization_url()
    # 请求令牌保存在会话中
    set_request_token(
        handler.request_token['oauth_token'],
        handler.request_token['oauth_token_secret'])
    # 重定向到认证 URL
    return redirect(auth_url)
```

```
@application.route('/callback')
def callback():
    """ 回调 URL
    Twitter 认证后的回调 URL
    """
    # 用 GET 方法获取 OAuth 的回调属性
    oauth_token = request.args.get('oauth_token')
    oauth_verifier = request.args.get('oauth_verifier')
    # 取消时重定向到首页
    if not oauth_token and not oauth_verifier:
        return redirect('/')
    # 获取 OAuth 的 handler
    handler = get_oauth_handler()
    # 从会话获取请求令牌并设置给 handler
    request_token = get_request_token(oauth_token)
    handler.request_token = {
        'oauth_token': oauth_token,
        'oauth_token_secret': request_token,
    }
    # 获取访问令牌
    access_token = handler.get_access_token(oauth_verifier)
    # 将访问令牌保存在会话中
    set_access_token(access_token)
    # 获取 API 实例
    api = get_api(access_token)
    # 获取登录用户的信息并保存至会话
    set_user(api.me())
    # 重定向到首页
    return redirect('/')

@application.route('/logout')
def logout():
    """ 登出 URL
    从会话中删除登录状态的信息
    """
    # 删除会话中的登录信息
    clear_session()
    # 重定向到首页
    return redirect('/')

@application.route('/timeline')
def timeline():
```

```
    """ 显示时间轴信息的页面
    需要认证
    """
    # 获取登录状态
    if not is_login():
        # 没有登录则重定向到首页
        return redirect('/')
    # 从会话获取访问令牌
    access_token = get_access_token()
    if not access_token:
        # 无法获取访问令牌时
        # 清除会话并重新开始
        clear_session()
        return redirect('/')
    # 获取 API 实例
    api = get_api(access_token)
    # 调用 Twitter API 获取时间轴信息
    statuses = api.home_timeline()
    # 通过模板显示页面
    return render_template('timeline.html', statuses=statuses)

if __name__ == '__main__':
    # 通过 IP 地址 127.0.0.1 的 5000 端口执行应用
    application.run('127.0.0.1', 5000, debug=True)
```

其中，CONSUMER_KEY、CONSUMER_SECRET 的值需要替换成之前为应用获取的值。在上面的例子中，我们用 Flask 的会话保存了请求令牌。该段代码使用了 index.html 和 timeline.html 两个模板文件，这两个文件保存在 templates 目录下，其内容如 LIST 15.41 所示。

LIST 15.41　templates/index.html

```
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <title>Twitter 认证 </title>
  </head>
  <body>
    {% if is_login %}
      <p> 登录名 : {{ user.screen_name }}</p>
      <ul>
        <li><a href="/timeline"> 获取时间轴信息 </a></li>
        <li><a href="/logout"> 登出 </a></li>
      </ul>
    {% else %}
```

```
      <ul>
        <li><a href="/login">登录</a></li>
      </ul>
    {% endif %}
  </body>
</html>
```

index.html 在未认证状态下显示开始认证处理的 URL 链接，在已认证状态下则显示 Twitter 的昵称（登录名）、获取时间轴信息的处理以及登出处理的链接（LIST 15.42）。

LIST 15.42　templates/timeline.html

```
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <title>时间轴信息的显示</title>
  </head>
  <body>
    <dl>
      {% for status in statuses %}
        <dt>{{ status.user.screen_name }}</dt>
        <dd>{{ status.text }}</dd>
      {% endfor %}
    </dl>
  </body>
</html>
```

timeline.html 用来显示获取到的时间轴信息（状态）。准备好源码及模板后，用 python 命令执行 `tweepy_auth.py` 启动服务器，代码如 LIST 15.43 所示。

LIST 15.43　用 python 命令执行 tweepy_auth.py

```
$ python tweepy_auth.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

NOTE

本例是用 5000 端口启动开发服务器的，因此请保证 SSH 隧道的主 OS 与客 OS 的 5000 端口相连接。

启动服务器后，打开 Web 浏览器试着访问 `http://127.0.0.1:5000/`。刚开始时我们处于未登录状态，所以会显示有登录链接的页面。点击登录连接，随后将跳转至 Twitter 的认证页面，我们会看到如图 15.9 所示的应用授权请求信息以及按钮。

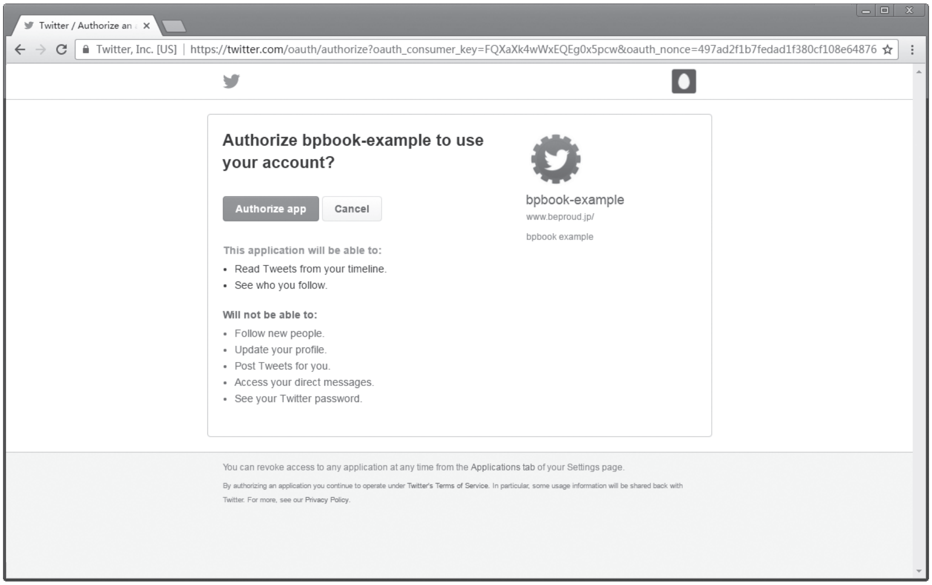


图 15.9　Twitter 的应用授权页面

点击页面中的授权按钮之后，页面将重定向到我们添加应用时填写的 Callback URL，处理也将返回到应用端。获取访问令牌和个人信息后，应用显示带有用户名、获取时间轴信息链接以及登出链接的登录状态界面。我们点击获取时间轴信息的链接之后，应用会使用用户的访问令牌获取时间轴信息并显示在页面上。至此，时间轴视图的应用开发完毕。

NOTE

这里介绍的时间轴视图是将访问令牌保存了在会话中，会话一旦过期就需要重新获取令牌。我们可以通过将访问令牌存放在数据库中来避免这一问题。

15.6　使用 REST API

系统间协作及使用外部服务时，经常会用 REST API 作接口。REST API 可以用 Python 标准模块 urllib 和 json 来控制，不过本书要讲的是 Requests 的使用方法。

Requests: HTTP for Humans
`http://docs.python-requests.org/en/latest/`

15.6.1　REST 简介

REST（Representational state transfer，具象状态传输[①]）是一种软件架构。它于 2000 年由 Roy Fielding 提出，是几种软件设计原则的集合。

不过，现在人们广泛使用的 REST 和 REST API 通常指“在 HTTP 上运行的非 SOAP 非 RPC 的 API”。

REST API 的特征如下。

- 在 HTTP 上运行
- 数据被称为“资源”
- REST API 可使用的资源具有唯一的 URL
- GET/POST/PUT/DELETE 等 HTTP 方法分别对应资源的获取 / 保存 / 覆盖 / 删除等操作
- 通过 JSON、XML 等格式收发数据
- 请求成功、请求失败等处理结果体现在状态代码中

如果要使用 REST API，则需要用到 HTTP 客户端。在 shell 命令或 shell 脚本中使用时要用 curl，在程序中使用时则要用 HTTP 客户端程序库。

REST - Wikipedia
　http://zh.wikipedia.org/wiki/REST
Fielding Dissertation CHAPTER 5 Representational State Transfer (REST)
　http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

15.6.2　导入 Requests

如 LIST 15.44 所示，用 pip 命令安装 Requests。本书使用的是 Requests 的 2.5.0 版本。

LIST 15.44　用 pip 命令安装 Requests

```
$ pip install requests
```

15.6.3　导入测试服务器

要检查 Requests 模块的运行状况就必须使用 Web 服务器。httpbin 模块是基于 Python 开发的 HTTP 测试服务器。本书将用 httpbin 模块作为测试服务器来讲解 Requests 的使用方法。

① 有时也被译为表述性状态转移或表述性状态传输等。——译者注

如 LIST 15.45 所示，用 pip 命令安装 httpbin。本书使用的是 httpbin 的 0.2.0 版本。

LIST 15.45　用 pip 命令安装 httpbin

```
$ pip install httpbin
```

httpbin 的测试服务器用 LIST 15.46 中的命令启动。

LIST 15.46　启动 httpbin 的测试服务器

```
$ python -m httpbin.core
 * Running on http://127.0.0.1:5000/
```

在导入 Requests 模块之前，先用 `curl` 命令检查测试服务器的运行状况。

在测试服务器已启动的状态下运行 LIST 15.47。如果运行正常，屏幕上将显示如下所示的 JSON 格式报告。

LIST 15.47　用 curl 命令检查测试服务器的运行状况

```
$ curl "http://127.0.0.1:5000/get?foo=bar"  # 向测试服务器发送 GET 请求
{
  "args": {  # args 为 GET 参数
    "foo": "bar"
  },
  "headers": {
    "Accept": "*/*",
    "Content-Length": "",
    "Content-Type": "",
    "Host": "127.0.0.1:5000",
    "User-Agent": "curl/7.35.0"
  },
  "origin": "127.0.0.1",
  "url": "http://127.0.0.1:5000/get?foo=bar"
}
$ curl -X post -d foo=bar http://127.0.0.1:5000/post  # 向测试服务器发送 POST 请求
{
  "args": {},
  "data": "",  # data 为发送数据的正文（未经过编码的表单数据除外）
  "files": {},  # files 为被上传的文件
  "form": {  # form 为经过编码的表单数据
    "foo": "bar"
  },
  "headers": {
    "Accept": "*/*",
    "Content-Length": "7",
```

```
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "127.0.0.1:5000",
    "User-Agent": "curl/7.35.0"
  },
  "json": null,
  "origin": "127.0.0.1",
  "url": "http://127.0.0.1:5000/post"
}
```

可以看出，测试服务器具有将请求的内容以 JSON 格式返回的功能。

NOTE

在 Windows 上使用 curl 命令时需要安装命令工具。

cURL - Download

http://curl.haxx.se/download.html

15.6.4　发送 GET 请求

用 Requests 发送 GET 请求，具体代码如 LIST 15.48 所示。

LIST 15.48　requests_get.py

```
# coding: utf-8
import pprint
import requests

def main():
    # GET 参数以字典形式通过 params 传值参数指定
    response = requests.get(
        'http://127.0.0.1:5000/get',
        params={'foo': 'bar'})
    # 使用响应对象的 json 方法可以获取转换为 Python 字典对象的 JSON 数据
    pprint.pprint(response.json())

if __name__ == '__main__':
    main()
```

在 requests.get 函数中指定对象 URL，用 params 关键字传值参数指定 GET 参数。服务器返回的响应为 JSON 格式，因此要用 json 方法将其转换为字典形式再显示到页面上。上述代码的执行结果如 LIST 15.49 所示。

LIST 15.49　执行结果

```
$ python requests_get.py
{u'args': {u'foo': u'bar'},
 u'headers': {u'Accept': u'*/*',
              u'Accept-Encoding': u'gzip, deflate',
              u'Connection': u'keep-alive',
              u'Content-Length': u'',
              u'Content-Type': u'',
              u'Host': u'127.0.0.1:5000',
              u'User-Agent': u'python-requests/2.5.0 CPython/2.7.8 Linux/3.13.
0-35-generic'},
 u'origin': u'127.0.0.1',
 u'url': u'http://127.0.0.1:5000/get?foo=bar'}
```

15.6.5　发送 POST 请求

用 Requests 发送 POST 请求，具体代码如 LIST 15.50 所示。

LIST 15.50　requests_post.py

```
# coding: utf-8
import pprint
import requests

def main():
    # POST 参数以字典形式通过第二个传值参数指定
    response = requests.post(
        'http://127.0.0.1:5000/post',
        {'foo': 'bar'})
    # 使用响应对象的 json 方法可以获取转换为 Python 字典对象的 JSON 数据
    pprint.pprint(response.json())

if __name__ == '__main__':
    main()
```

request.post 函数中指定了对象 URL 和 POST 参数。给 POST 参数指定的字典会被编码为 URL 发送出去。这段代码的执行结果如 LIST 15.51 所示。

LIST 15.51　执行结果

```
$ python requests_post.py
{u'args': {},
 u'data': u'',
```

```
 u'files': {},
 u'form': {u'foo': u'bar'},
 u'headers': {u'Accept': u'*/*',
              u'Accept-Encoding': u'gzip, deflate',
              u'Connection': u'keep-alive',
              u'Content-Length': u'7',
              u'Content-Type': u'application/x-www-form-urlencoded',
              u'Host': u'127.0.0.1:5000',
              u'User-Agent': u'python-requests/2.5.0 CPython/2.7.8 Linux/3.13.
0-35-generic'},
 u'json': None,
 u'origin': u'127.0.0.1',
 u'url': u'http://127.0.0.1:5000/post'}
```

15.6.6　发送 JSON 格式的 POST 请求

有些 API 接口要求直接发送 JSON 格式的字符串，不能将数据编码为 URL。用 Requests 以 JSON 格式字符串的形式发送 POST 请求，具体代码如 LIST 15.52 所示。

LIST 15.52　requests_post_json.py

```
# coding: utf-8
import pprint
import json
import requests

def main():
    # 指定 json.dumps 生成的字符串之后，可以直接发送数据而不进行 URL 编码
    # 需要明确指定 Content-Tpye
    response = requests.post(
        'http://127.0.0.1:5000/post',
        json.dumps({'foo': 'bar'}),
        headers={'Content-Type': 'application/json'})
    pprint.pprint(response.json())

if __name__ == '__main__':
    main()
```

post 函数的传值参数中指定的发送数据为字符串。传值参数指定字符串时会直接发送数据，不进行 URL 编码。这段代码的执行结果如 LIST 15.53 所示。

LIST 15.53　执行结果

```
$ python requests_post_json.py
{u'args': {},
 u'data': u'{"foo": "bar"}',
 u'files': {},
 u'form': {},
 u'headers': {u'Accept': u'*/*',
              u'Accept-Encoding': u'gzip, deflate',
              u'Connection': u'keep-alive',
              u'Content-Length': u'14',
              u'Content-Type': u'application/json',
              u'Host': u'127.0.0.1:5000',
              u'User-Agent': u'python-requests/2.5.0 CPython/2.7.8 Linux/3.13.
0-35-generic'},
 u'json': {u'foo': u'bar'},
 u'origin': u'127.0.0.1',
 u'url': u'http://127.0.0.1:5000/post'}
```

15.6.7　使用 GET/POST 之外的 HTTP 方法

某些 API 还会用到 GET、POST 之外的 HTTP 方法（PUT、DELETE、HEAD、OPTIONS）。Requests 同样为这些方法准备了对应的函数，代码如 LIST 15.54 所示。

LIST 15.54　requests_other_methods.py

```
# coding: utf-8
import requests

def main():
    requests.put('http://127.0.0.1:5000/put', {'foo': 'bar'})  # PUT
    requests.delete('http://127.0.0.1:5000/delete')  # DELETE
    requests.head('http://127.0.0.1:5000/get')  # HEAD
    requests.options('http://127.0.0.1:5000/options')  # OPTIONS

if __name__ == '__main__':
    main()
```

执行了这段代码之后，程序会向服务器发送 HTTP 的 PUT、DELETE、HEAD、OPTIONS 方法的请求，获取响应后执行结束。

可见，用 Requests 可以轻松使用以 HTTP 为接口的 REST API。

15.7 小结

在开发应用的过程中，我们多少都会遇到些难题，而方便好用的模块往往是解决难题的方法之一。本章介绍了几个开发中的难题以及解决相应难题的 Python 模块。希望各位多多利用本书以及其他各种信息来源，收集难题的解决方案以及好用模块的信息。多了解一些可以拿来就用的模块是大幅缩短开发时间的有效途径。

附录

附录 A　VirtualBox 的设置

近年来，PC 的虚拟化技术不断发展，个人已经能够轻松地搭建并使用虚拟机了。随着虚拟机个人使用的简化，越来越多的开发者也开始利用虚拟化软件搭建本地开发环境。这里我们学习一下将 Oracle 的 VirtualBox 设置为虚拟机的流程。VirtualBox 是一款支持 Windows、OS X、Linux 的免费虚拟化软件。

这里我们将本地计算机称为主 OS，虚拟机称为客 OS。主 OS 端的系统以 OS X 为例进行说明。

A.1　安装 VirtualBox

首先从 VirtualBox 的下载页面[①] 下载适合主 OS 的 VirtualBox。本书使用了编写时（2014 年 11 月）的最新版本 VirtualBox 4.3.18（图 A.1）。

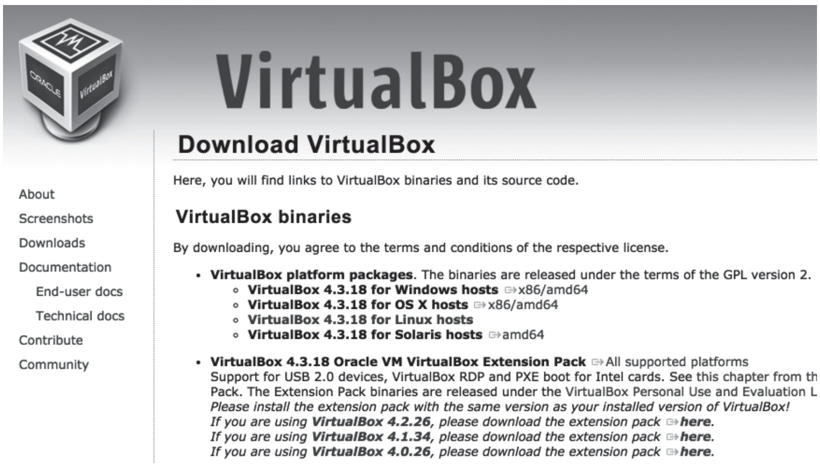


图 A.1　下载 VirtualBox

下载完成之后按照各 OS 的安装向导进行安装。出现图 A.2 所示的启动界面时，表示 VirtualBox 安装完毕。

① `https://www.virtualbox.org/wiki/Downloads`

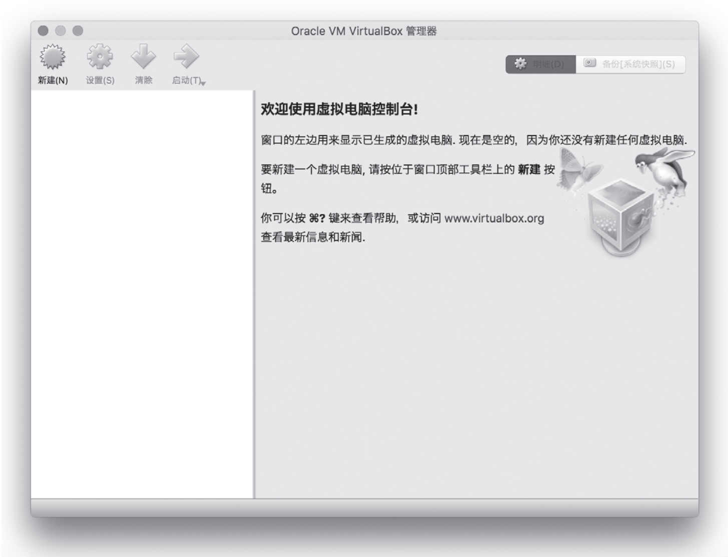


图 A.2　VirtualBox 的启动界面

A.2　新建虚拟机

接下来在 VirtualBox 上新建虚拟机。点击 VirtualBox 的“新建”按钮（图 A.3）。



图 A.3　创建虚拟机

本例中的 OS 类型的配置如下。

- 操作系统：Linux
- 版本：Ubuntu（64 bit）

图 A.3 中设置了 64 bit 版的 Ubuntu，各位可按照自己的客 OS 进行更改。VirtualBox 的版本有 Ubuntu 和 Ubuntu（64 bit）可选。点击下一步之后系统会询问下述内容，各位请按照自己的环境进行设置。

- 内存
- 虚拟硬盘
- 虚拟盘的文件类型
- 虚拟盘的容量

设置完成后，一个空的虚拟机镜像就创建好了（图 A.4）。



图 A.4　虚拟机新建完毕

A.3　备份虚拟机

在阅读本书的过程中，各位会遇到许多服务器设置的变更。在修改设置时，难免出现设置错误甚至设置损坏，导致虚拟机无法使用。为防备这种情况发生，我们要为当前已设置完毕的虚拟机做好备份，以便出了问题能从头再来。

接下来学习一下如何备份虚拟机以及如何从备份还原。使用 VirtualBox 备份虚拟机的方法

有许多，这里我们选其中比较简单的“虚拟电脑的导出与导入”进行学习。

这里提到的虚拟电脑是对包含设置在内的所有文件进行备份的机制。该机制将备份数据以 OVA（Open Virtualization Format Archive）文件的形式保存，当我们的虚拟机出现问题时，只要有这个文件，就可以在 VirtualBox 上将虚拟机恢复到导出时的状态。如果只是想临时保存虚拟机的状态，还可以使用 VirtualBox 的“快照”功能。

备份前先关闭客 OS。关闭后依次选择“VirtualBox > 管理 > 导出虚拟电脑”。点击导出虚拟电脑后会出现图 A.5 所示的界面，我们可以在该界面中选择虚拟机。选择完毕后点击下一步。



图 A.5　导出虚拟电脑（1）

接下来选择导出到的位置，再点击下一步开始导出。



图 A.6　导出虚拟电脑（2）

这样，我们就完成了虚拟电脑的导出（图 A.6）。

导入也很简单，只要依次选择“VirtualBox > 管理 > 导入虚拟电脑”，选择之前导出的 ova 文件进行导入，我们就又能使用该设置下的虚拟机了。VirtulBox 的设置就了解到这里。接下来，在附录 B 中，我们将学习如何在新建的虚拟机镜像中安装 OS（Ubuntu）。

附录 B　OS（Ubuntu）的设置

我们在附录 A 中已经准备好了环境，现在来了解一下安装客 OS（Ubuntu）的流程。

B.1　安装 Ubuntu

首先从下述 URL 下载 Ubuntu 的 CD 镜像（图 B.1）。Ubuntu 为多种执行环境准备了相应的镜像，本书使用的是 Server 版。

```
http://www.ubuntu.com/download/server
```

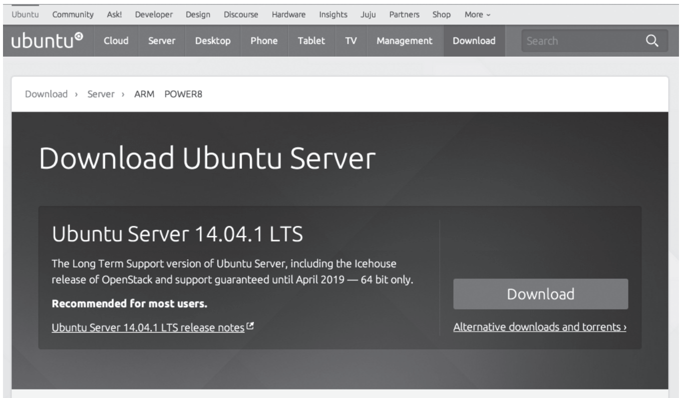


图 B.1　下载 Ubuntu

这里我们使用的版本是 ubuntu-14.04.1-server-amd64.iso。下载完成之后启动之前创建好的虚拟机。初次启动时会显示如图 B.2 所示的安装媒体选择界面。

这里选择刚刚下载的 CD 镜像并继续。之后界面上会显示启动按钮，点击按钮实际启动 OS 即可。



图 B.2　VirtualBox 初次启动窗口

安装包启动后，进行 Ubuntu 的安装设置。

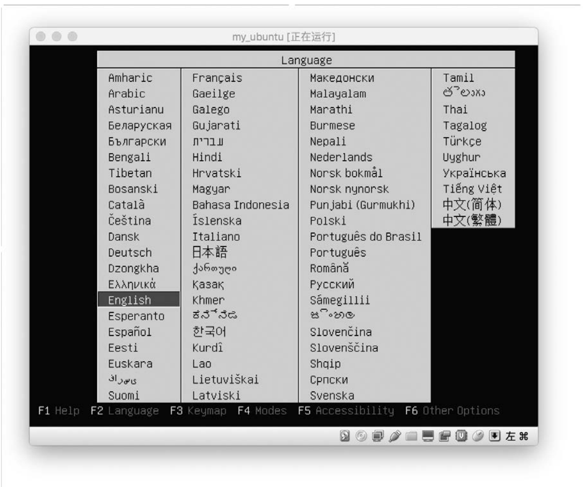


图 B.3　Ubuntu 的安装界面

首先是语言设置，最开始请选择 English 并按 Enter 键（图 B.3）。一开始选择中文会导致在 VirtualBox 上运行的控制台显示乱码，所以语言问题等到后面再作调整。



图 B.4　Ubuntu 的安装准备

设置完语言之后选择 Install Ubuntu Server 进入下一步（图 B.4）。



图 B.5　Ubuntu Select a language

如图 B.6 ~ 图 B.8 所示，安装开始后，系统将询问“选择当前所在地”和“键盘设置”，各位请根据自己的情况进行设置。

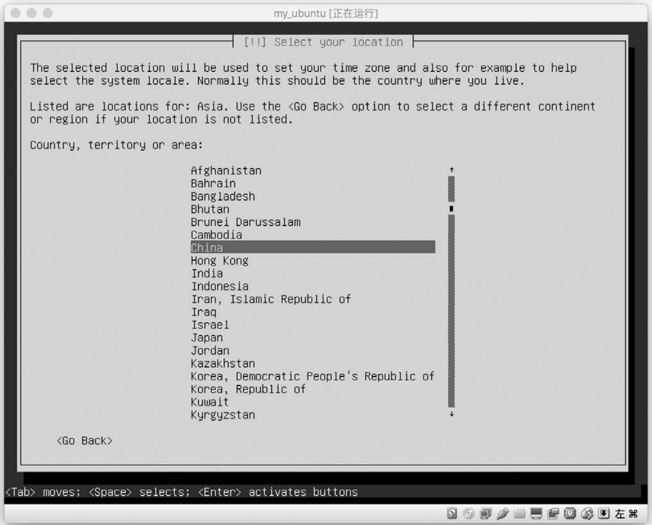


图 B.6　Ubuntu Select your location



图 B.7　Ubuntu Configure locales

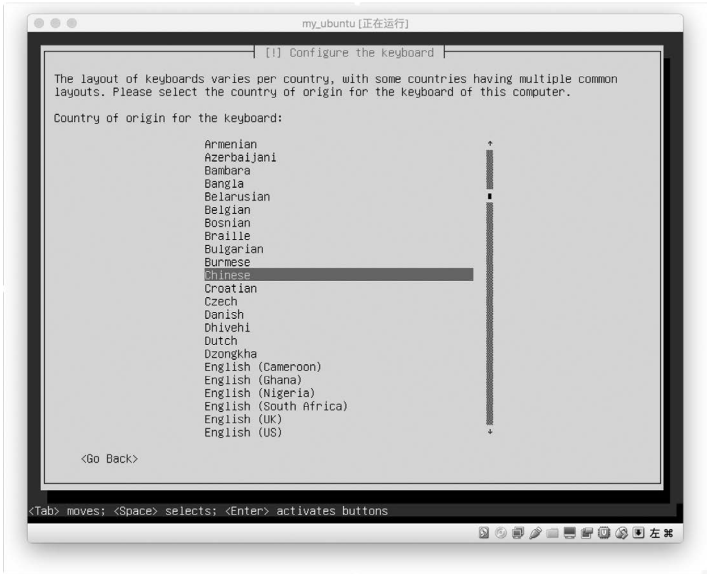


图 B.8　Ubuntu Configure the keyboard

设置完成后，安装就会正式开始。接下来会显示网络的设置。如图 B.9 所示输入 Hostname 并进入下一步。

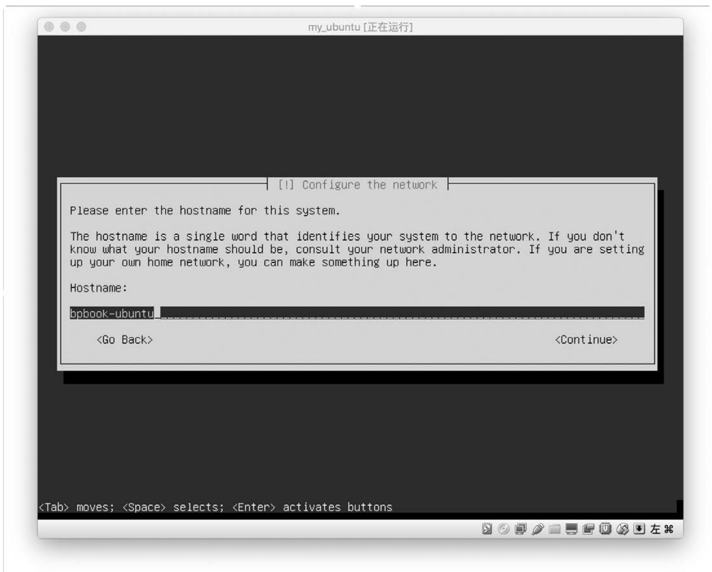


图 B.9　Ubuntu Configure the network

根据安装进度，接下来会进入创建账户的界面，这里要设置“用户名”和“密码”。这里我

们设置一个名为 bpbook 的用户（图 B.10）。

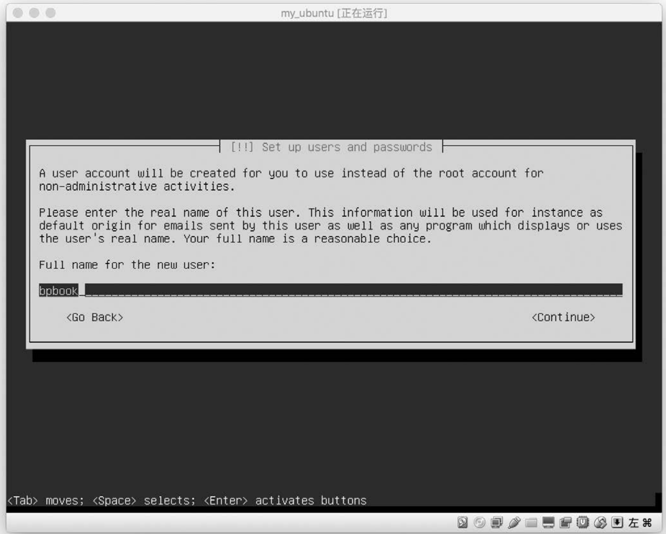


图 B.10　Ubuntu Set up users and passwords

接下来是时区的设置，系统会自动为我们推测当前时区。确认无误后点击 Yes（图 B.11）。

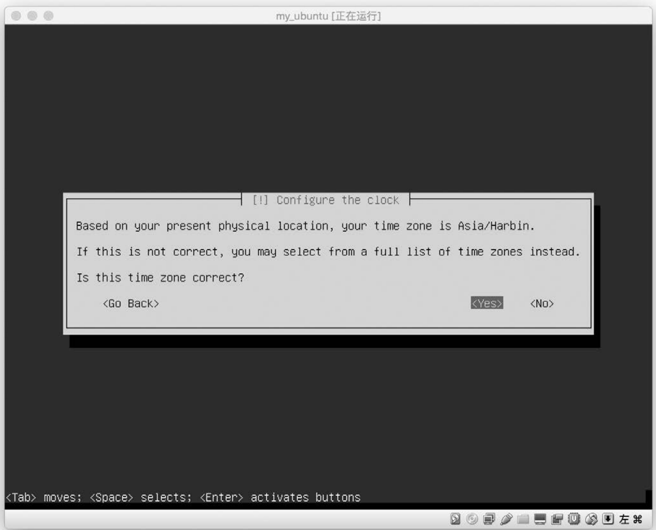


图 B.11　Ubuntu Configure the clock

然后是硬盘分区的设置。选择 Guided - use entire disk and set up LVM 并继续（图 B.12）。

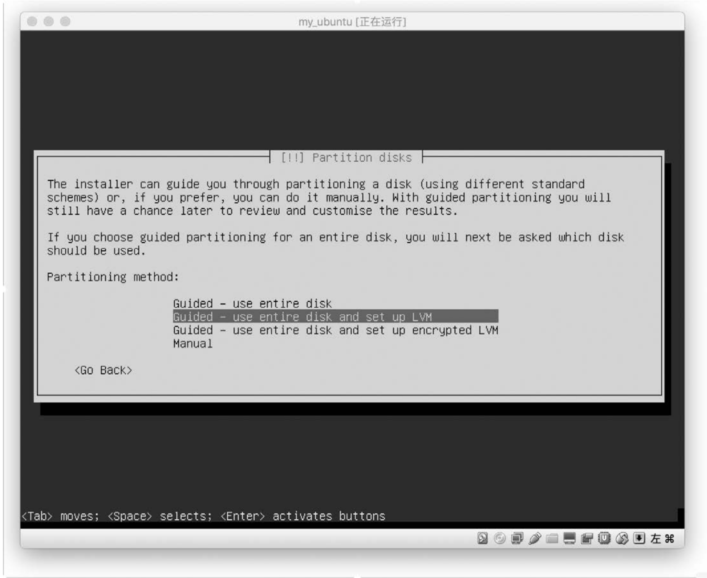


图 B.12　Ubuntu Partition disks（1）

随后的分区设置会询问 Write the changes to disk and configure LVM，这里选择 Yes（图 B.13）。随后还会出现几次询问，基本上都选 Yes 就可以了。

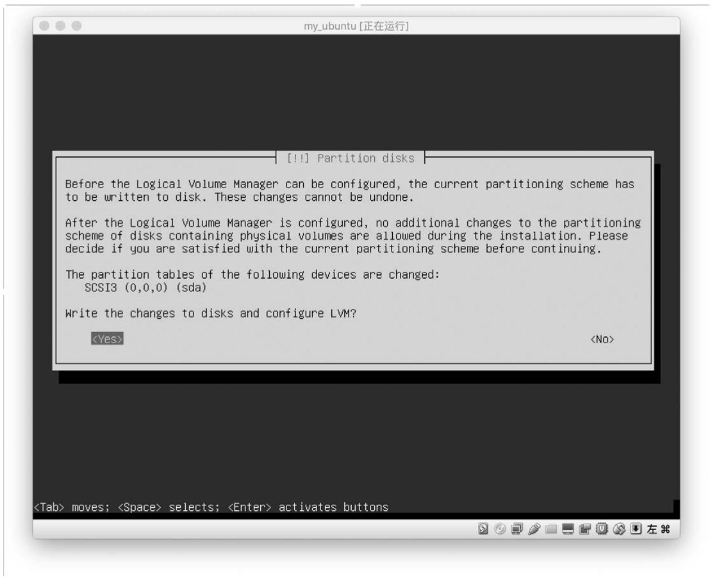


图 B.13　Ubuntu Partition disks（2）

如果是网络连接受限的环境，建议设置 HTTP 代理以保证程序包管理器可以访问网络。不

受限制的情况下则可以直接选择 Continue 继续安装（图 B.14）。

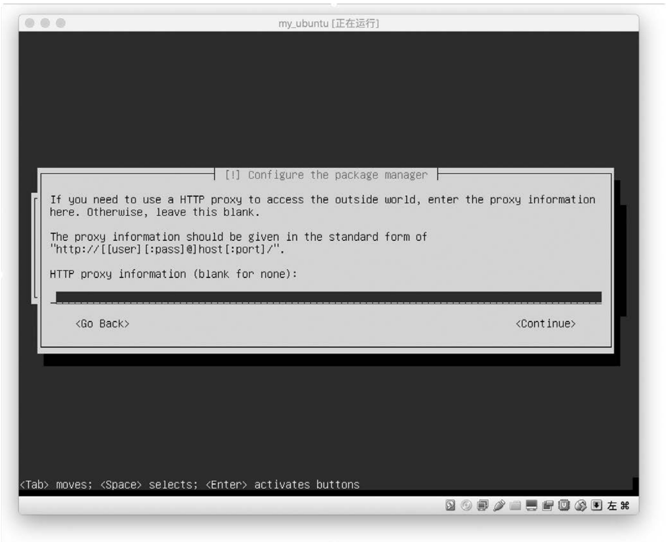


图 B.14　Ubuntu Configure the package manager

与安全更新相关的问题要选择 Install security updates automatically（图 B.15）。OS 开发每天都在进行，其中包括很多重要的安全更新，因此我们需要选择这个选项，以保证 OS 能持续安全运行。

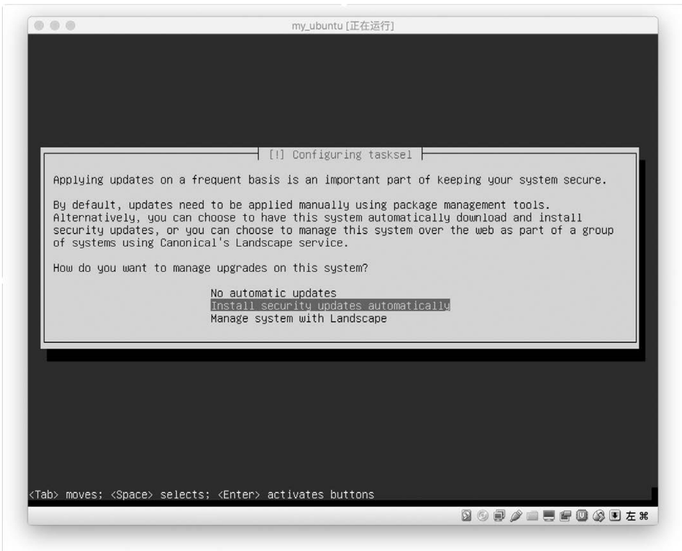


图 B.15　Ubuntu Configuring tasksel

由于 Ubuntu Server 版是精简安装，所以这个阶段并没有安装任何软件。这里我们只勾选 OpenSSH server 并继续（图 B.16）。选择由空格键完成。

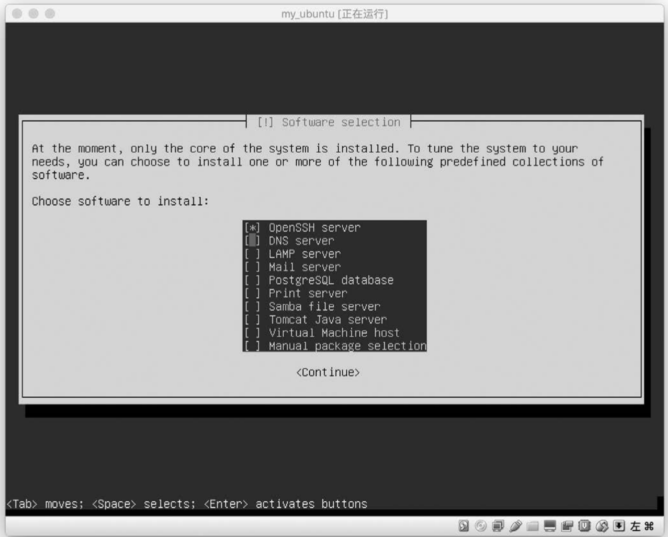


图 B.16　Ubuntu Software Selection

最后会询问是否安装 GRUB 引导加载程序，这里选择 Yes 并继续（图 B.17）。

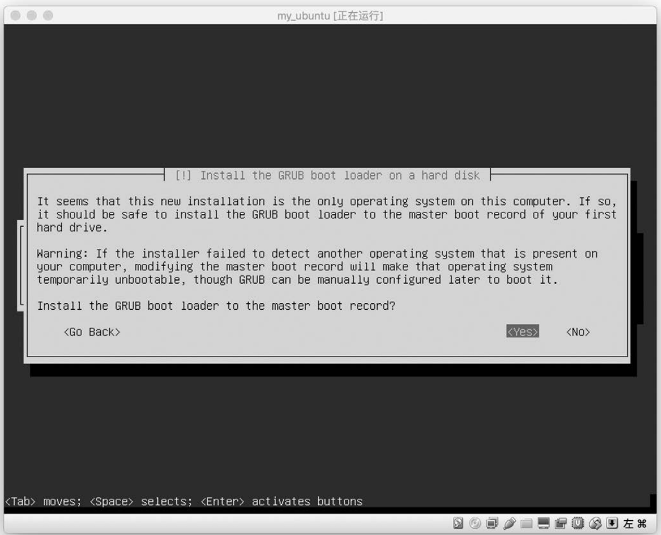


图 B.17　Ubuntu Install the GRUB boot loader on a hard disk

之后只要等安装进度条走完，我们就完成客 OS 的安装了。安装完毕之后会要求重启，所以我们这里立刻重新启动（图 B.18）。

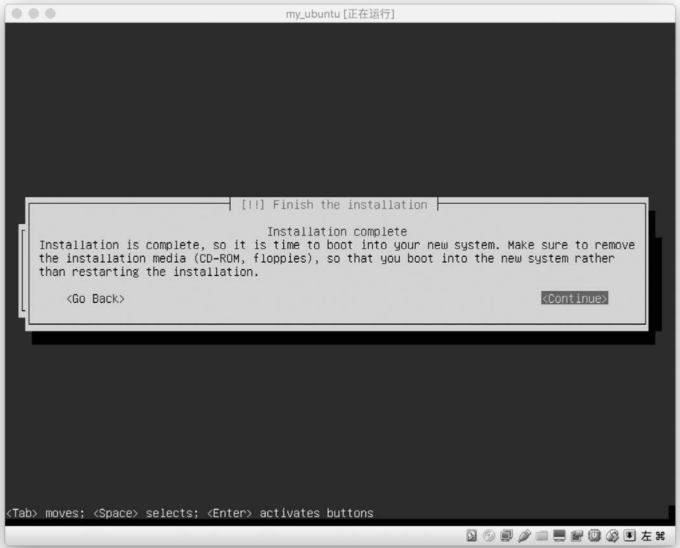


图 B.18　Ubuntu Finish the installation

客 OS 重启后将显示如图 B.19 所示的登录界面。用前面创建的用户登录即可。

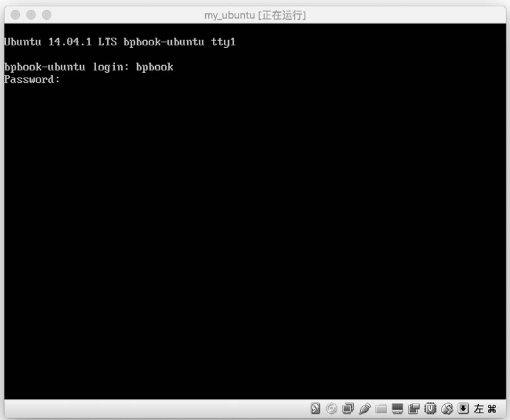


图 B.19　Ubuntu 的登录界面

B.2　SSH 的设置

接下来要让主 OS 通过 SSH 连接虚拟机上的客 OS。我们在图 B.16 所示的步骤中已经安装了 OpenSSH server，因此不必再另行安装了。现在在客 OS 的控制台执行下述命令，检查 ssh 模块是否已经启动（LIST 1）。

LIST 1　查看进程

```
$ ps aux | grep sshd
```

如果进程列表中有 /usr/sbin/sshd -D，就表示 SSH 守护进程已经启动。

NOTE

如果前面忘记安装 OpenSSH server，则需要通过 `apt-get` 等命令手动安装。

```
$ sudo apt-get install ssh
```

现在 SSH 已经准备就绪，可以从主 OS 通过 SSH 登录客 OS 了。VirtualBox 提供了多种连接方法，这里我们用 NAT 连接的端口转发机制，将主 OS 的 2222 端口与客 OS 的 SSH 端口（22）连接起来。

从 VirtualBox 管理界面选择“设置 > 网络 > 网卡 1 > 端口转发”，然后如图 B.20 进行设置。



图 B.20　VirtualBox 端口转发的设置

如图 B.20 设置完毕后，在主 OS 的控制台上输入 LIST 2 所示的命令。

LIST 2　SSH 连接

```
$ ssh -p 2222 bpbook@127.0.0.1
The authenticity of host '[127.0.0.1]:2222 ([127.0.0.1]:2222)' can't be established.
RSA key fingerprint is d0:e8:9f:96:8e:24:0d:96:dc:0b:51:fb:7f:b7:1b:f0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[127.0.0.1]:2222' (RSA) to the list of known hosts.
bpbook@127.0.0.1's password:
```

NOTE

这里假定主 OS 为 OS X，因此使用的是 ssh 命令。而 Windows 没有 ssh 命令，因此需要使用其他软件（例如 PuTTY①）进行 SSH 连接。

@ 前的 bpbook 是安装 Ubuntu 时创建的用户名，各位请替换成自己的用户名。初次连接时系统会询问 Are you sure you want to continue connecting (yes/no)?，这里输入 yes。随后系统还会要求输入密码，此时输入创建用户时设置的密码即可。顺利登录客 OS 后将显示图 B.21 所示的内容。

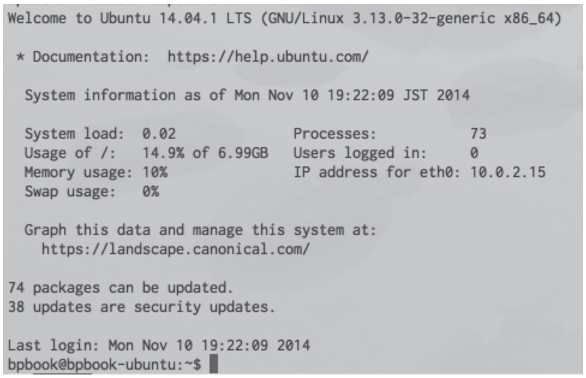


图 B.21　主 OS 通过 SSH 连接客 OS

NOTE

一般说来，通过 SSH 连接服务器时只校验密码并不安全。本节内容面向的是个人开发环境的搭建，因此只讲了用密码登陆服务器的方法。要想搭建更加安全的环境，建议使用“公钥加密系统”进行认证。

① http://www.chiark.greenend.org.uk/~sgtatham/putty/

公钥加密 https://zh.wikipedia.org/wiki/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5%E5%8A%A0%E5%AF%86

B.3　中文的设置

LIST 3　安装中文语言包

```
sudo apt-get install language-pack-zh-hans
sudo update-locale LANG=zh_CN.UTF-8
```

执行 LIST 3 中的命令后即可安装中文的 locale 文件并完成设置。安装完成之后在 login-shell 的设置文件（.bashrc）中设置下述环境变量（LIST 4）。

LIST 4　在 .bashrc 中设置用于设定语言的环境变量

```
export LANG="zh_CN.UTF-8"
export LANGUAGE="zh_CN:zh"
```

设置完成之后，用 source 命令反映设置（LIST 5）。

LIST 5　反映 .bashrc 的设置

```
$ source ~/.bashrc
```

到这里，中文就设置完毕了。以后再启动支持中文的 Vim 等程序时，可以看到信息都是以中文显示的。

B.4　添加用户

用于开发的服务器上常要运行许多应用，但从安全角度讲，最好不要用 root 用户来运行它们。

所以，我们需要根据应用的开发进度，设置能够通过 sudo 命令修改服务器和中间件设置的组以及用户。这里我们以添加 dev 组和 bpuser 用户为例进行学习。

首先用 bpbook 用户登录并创建组。如 LIST 6 所示，以 root 权限执行 groupadd 命令来创建 dev 组。

LIST 6　添加组

```
$ sudo groupadd dev
```

接下来创建用户。用 adduser 命令创建 bpuser 用户。此时用 --ingroup 选项指定 dev 可以让新用户加入 dev 组（LIST 7）。

LIST 7　添加用户

```
$ sudo adduser bpuser --ingroup dev
```

这样一来就创建了 bpuser 用户。需要删除用户时使用 userdel 命令进行删除（LIST 8）。

LIST 8　删除用户

```
$ sudo userdel -r bpuser
```

Ubuntu 14.04 默认有 sudo 组，被添加到这个组的用户均可以使用 sudo 命令。

LIST 9　向组添加用户

```
$ sudo usermod -aG sudo bpuser
```

不过在目前的设置下，dev 组的用户每次使用 sudo 命令都要输一遍密码，这必然影响开发环境的便捷性。因此我们要让 dev 组的用户在执行 sudo 命令时免去输入密码的麻烦。

我们在 /etc/sudoers.d/ 目录下创建一个文件，并给 dev 组设置权限。先执行下述 visudo 命令（LIST 10）。

LIST 10　执行 visudo

```
$ sudo visudo -f /etc/sudoers.d/dev
```

NOTE

对 sudo 进行个别设置时，尽量不要编辑原设置文件 /etc/sudoers，最好是如上所述在 /etc/sudoers.d/ 下放置设置文件。因为在升级 OS 等时容易无意中覆盖掉 /etc/sudoers 文件，导致所有账户都无法使用 sudo 命令。这样做可以免除上述顾虑。

执行 visudo 命令后会生成 /etc/sudoers.d/dev 文件。对该文件作如下描述。

LIST 11　sudo 设置示例

```
%dev ALL=(ALL) NOPASSWD:ALL
```

按照 LIST 11 所示设置之后，dev 组的所有用户就都可以不输入密码直接执行 sudo 命令了。

NOTE

本节，我们站在本地开发环境的角度，学习了如何设置无需密码执行 sudo 命令，以维持开发环境的便捷性。

但是，sudo 命令可以使用 root 权限对服务器进行操作，一旦用户被窃取，带来的损害将会非常大。这个风险我们一定要清楚。

因此，如果我们使用的是可通过互联网访问的共享服务器、正式运营的服务器，那么当我们考虑设置“无需密码执行 sudo 命令”时，一定要慎之又慎。

作者介绍

清水川贵之

执笔第 3 章、第 7 章、第 9 章。2011 年 6 月入职 BePROUD。曾参与 Zope2/3 的应用开发、Python 认证服务器的实现、Ruby on Rails 业务系统的开发。2003 年起开始参与自动测试以及持续集成的环境搭建等工作，同时对 zc.buildout 等技术的运用深有研究。文档编辑工具 Sphinx 的 Committer、Sphinx-users.jp 运营人员、一般社团法人 PyCon JP 理事。著作及译作有《Sphinx 入门》[①]、《Python 高级编程》[②]。

Twitter: @shimizukawa

URL: http:// 清水川 .jp/

冈野真也

执笔第 2 章、第 6 章、第 12 章、第 15 章。2008 年 8 月入职 BePROUD。从 Django 框架 0.95 版时代起就开始使用该框架，同时参与了该框架的日文翻译工作。工作方面多担任设计、开发以及技术顾问。

Twitter: @tokibito

hatena: id:nullpobug

drillbits

执笔第 1 章、第 11 章。参与 Python 开发之前还参与过 COBOL、Java、ActionScript 语言的开发，近来正在全心全意进行 JavaScript 开发。在自己家里录动画时会用 Python。

Twitter: @drillbits

URL: http://drillbits.jp/

cactusman

执笔第 10 章。BePROUD 员工、日本 Jenkins 用户协会干事 &Jenkins Committer。在研究

---

① 原书名为『Sphinx をはじめよう』(2013 オライリー・ジャパン刊)，暂无中文版。——编者注

② 原书名为 *Expert Python Programming*，Tarek Ziadé 著，中文版名为《Python 高级编程》，姚军等译，人民邮电出版社，2010 年 1 月。——编者注

XP 之一 CI 时接触到 Jenkins，被其制作之精良深深折服。兴趣是一切游戏（包括桌游）。

Twitter: @cactusman

hatena: id:cactusman

东健太

执笔第 5 章、第 11 章。2008 年职高毕业，经手机应用供应商介绍入职 BePROUD。业务从开发到运营均有涉猎，近来多负责基础设置的搭建。在本书上一版中负责的章节内容已陈旧过时，只得彻底重写。

Twitter: @feiz

hatena: id:feiz

tell-k

执笔第 1 章、第 12 章、附录。2011 年 4 月入职 BePROUD。从事 Python 编程相关工作。

Twitter: @tell-k

hatena: id:tell-k

文殊堂

执笔第 6 章。2009 年 1 月入职 BePROUD。从事 JavaScript 跨平台 GUI 的开发工作。

Twitter: @monjudoh

hatena: id:monjudoh

富田洋佑

执笔第 4 章、第 5 章。2011 年 12 月入职 BePROUD。曾混迹于快餐业、游戏业，后以经理身份加入 BePROUD。

Twitter: @you_tomita

aodag

执笔第 8 章、第 9 章。BePROUD 员工。曾在业务型公司、Web 服务、影视开发领域任职，后加入 BePROUD。从 1.5 版本开始迷上 Python，如今视 Python 如左膀右臂。最近致力于 Pyramid 投稿与封装的启蒙教育工作。

Twitter: @aodag

铃木 Takanori

执笔第 4 章、第 5 章。2012 年 3 月入职 BePROUD。在之前的单位开发内部网站时接触了 Zope/Plone，随后因开发所需开始使用 Python。目前主要活动有任 PyCon JP 2014 和 2015 届主席、一般社团法人 PyCon JP 理事、Python 攀岩社团（#kabepy）团长，并主办 Python mini Hack-a-thon、参加 Plone User's Group Japan。合著有《Plone 完全应用指南》[①]、《Plone 4 Book》[②]。

Twitter: @takanory

URL: http://takanory.net/

清原弘贵

执笔第 12 章、第 14 章。2012 年 10 月入职 BePROUD。进入 BePROUD 之前曾是一名客户工程师。喜爱 Django，常出于兴趣开发一些 Web 应用或程序库，有时还给已有代码打打补丁。对于 Django 贡献者名单中有自己的名字一事感到十分自豪。

Twitter: @hirokiky

URL: http://hirokiky.org/

《Python 开发实战》[③] 执笔者

清水川贵之、冈野真也、池田洋介、畠弥峰、drillbits、cactusman、东健太、tell-k、今川馆、Natsu、文殊堂、aita、冨田洋佑

① 原书名为『Plone 完全活用ガイド』（2008 技術評論社刊），暂无中文版。——编者注

② 原书名为『Plone 4 Book』（2011 Talpa-Tech 刊），暂无中文版。——编者注

③ 即本书上一版本，2014 年 6 月人民邮电出版社出版。——编者注

日本极客和书虫们的智慧结晶和经验总结

BePROUD里不乏极客和书虫们。在这里，很多人对特定领域的了解程度能吓掉你的下巴。大家一旦发现感兴趣的事，就会拿出私人时间来学习、实践。要知道，极客和书虫们不会为这种事情吝啬时间。

正如人们印象中的那样，极客和书虫们大多有些怪癖，但BePROUD的员工都具备下列共识。

- **希望能不做不想做的事**
- **希望学会好的方法并付诸实践**
- **希望工作时有个好心情**

本书的内容全部基于事实，都是BePROUD员工实际尝试、实践过的。我们希望给各位提供一些能实际应用且行之有效的知识，而不是让各位去死记硬背一大堆晦涩难懂的概念。我们很愿意看到本书的知识能对各位有所帮助，愿各位能在工作中有个好心情。

——摘自引言

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/程序设计
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43856-0
定价：79.00元